Monitoring & Remote Operation of an Engine Test Cell

MONITORING & REMOTE OPERATION OF AN ENGINE TEST CELL

BY

JAMIE TURNER, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2014) McMaster University

(Department of Computing & Software) Hamilton, Ontario, Canada

TITLE:	Monitoring & Remote Operation of an Engine Test Cell
AUTHOR:	Jamie Turner B.Eng., (Mechatronic Engineering) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Martin von Mohrenschildt
NUMBER OF PAGES:	xiii, 87

To my family, friends, and colleagues.

Abstract

In the automotive industry engines are regularly tested and evaluated by running them for a prolonged time under controlled conditions; environmental conditions, engine load, and drive cycle. These tests are performed in an engine test cell; a computer controlled environment with mechanical fittings and sensors to facilitate the testing of an engine.

Our goal was to develop a software suite that provides a distributed graphical interface to the data acquisition and control systems of an engine cell. As we found existing systems to be inadequate in providing a distributed interface, we designed and developed a light weight flexible software suite to remotely, over a network, observe and control the parameters in an engine cell. We used the Fast Light Toolkit (FLTK) GUI library, with networking sockets and process threads to establish the software architecture of the engine test system.

Through use of process threads, the client architecture divides tasks into network data sending and receiving, local channel synchronization, and interface operation. Networking sockets used in network data sending and receiving facilitate synchronization of each clients' channel storage and host's channel data. The FLTK GUI library produces visual interactive components of the interface for invoking interactions.

Distributed interfacing allows display and modification of the engine cell's operation remotely in locations where relocating an engine cell is not feasible. These locations, such as demonstrations to distant clients and meeting rooms, display the current status of the engine cell through its interfaces without requiring migration of the engine cell to the specified rooms.

Acknowledgements

I would like to give my thanks to my fellow workers, for all the long hours they have given to helping me write and defend my thesis. I would like to thank my family, for the support they have given me over the years for my endeavours. To my colleagues, I give them my best for their efforts and the information they have provided to let this project reach what it is now.

Notation and abbreviations

Acronym	Meaning
DAQ	Data Acquisition
DAQC	Data Acquisition and Control
ESTOP	Emergency Stop
ETC	Engine Test Cell
ETCS	Engine Test Cell Suite
FLTK	Fast Light Toolkit
FLUID	Fast Light User Interface Designer
GUI	Graphical User Interface
OOP	Object-Oriented Programming
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
XML	Extensible Markup Language

Any abbreviations missing from here will be explained and expanded on within the document.

Contents

List of Figures

Chapter 1

Introduction

Figure 1.1: Automotive Engine Test Cell
(Centre for Mechatronics and Hybrid Technologies, 2013)

In the automotive industry, testing is a major task in ensuring vehicle reliability. Engineers have many advanced tools available to assist them in facilitating testing. A common tool used for testing engines is the *engine test cell*. An *engine test cell* contains the following components:

- Engine
- Dynamometer
- Transducers
- Data acquisition & control unit
- Host

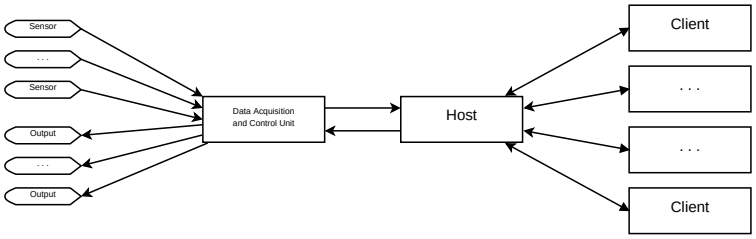

Figure 1.2: Layout of *engine test cell*

The purpose of the *engine test cell* is to create an environment where quantifiable data can be acquired from the *engine* and used on a *host* computer. Data acquired from an *engine* are quantities pertaining to linear or rotational movements of mechanical parts such as the crankshaft or pistons, electrical voltages and currents such as spark plugs and alternator outputs, or thermal/barometric levels at different points on the *engine*.

In the *engine test cell*, we were previously using LabVIEW as its monitoring and control interface. Unfortunately LabVIEW is closed source, which limited our flexibility in research. In order to improve control of the *engine test cell* beyond the range of LabVIEW, we created our own interface suite. This project aimed at developing a solution to observing and interacting with the *engine test cell*. We needed a lightweight product and less restrictive license for use in observation and research of the engine. To achieve this task, we decided to create a new *suite* as an alternative to LabVIEW's virtual instrument in use.

A *suite* is a collection of software tools and modules used to accomplish tasks such as:

- Gathering and pruning of data from outputs
- Mathematical calculations/modifications on sets of data
- Output data into human-readable format(sending to displays and other devices)
- Output data into machine-readable format(sending to other suites/systems)

The project was further extended to allow the *host* to send data to *client* computers. These *clients* connect to the *host* to retrieve data and send commands to observe and alter the *engine's* set points. The main focus of the project was to create both the *host* and the *client* to collaboratively monitor and control the engine cell.

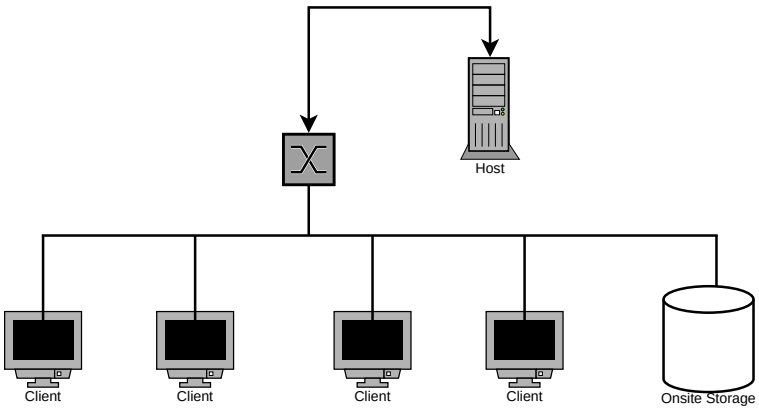

Figure 1.3: Layout of *host* and *client*

In isolating the graphical user interface to the *clients*, we separate the observation and controlling aspects from the *data acquisition unit* and *host* to the *clients*, making it more economical to reduce the processing requirements of the *host* to a smaller, more portable computer system. The purpose of this was to separate the project into smaller sections for deploying the DAQC compactly into a vehicle or other such environment.

To make the system operate in a specified manner, tuning must be done beforehand. Tuning the system is done to set up how feedback loops and set points will affect the operation of the *engine test cell suite*. This tuning is crucial for crafting a system to do what is desired. In the *engine test cell suite*, the *host* can be designed to accomplish feedback through deriving output channels from specified input channels. As each channel can be linked to inputs from the *clients*, modification of the *host's* feedback loop parameters can be changed during execution if desired by the *engine test cell suite* designer.

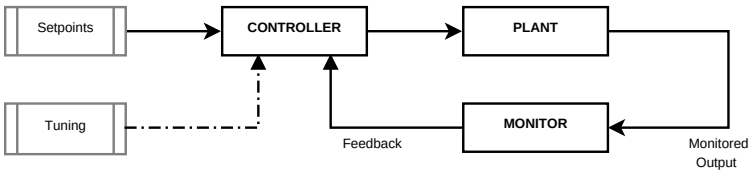

Figure 1.4: System Feedback Loop

When monitoring and/or controlling a system, there must be a method of communicating between the monitoring/*set point* (*host*) system and the target controlling (DAQC) system. If the two systems are physically close in proximity to each other, it may only be necessary to physically connect the control wires (if electrical), or drivechains and pumps (if mechanical). In cases where being close to one another is not possible, remote connectivity over networked communications may be required. Networked communications are done with software and hardware to allow transmitting and receiving information compactly over communication lines such as:

- Telephone Lines (RJ-11, RJ-23)
- Serial Communication (I2C, SPI, TWI)
- Ethernet Connections (RJ-45)
- Proprietary Connectors

To interface with the *engine test cell suite*, a *user interface* (UI) is built to facilitate operations. *User interfaces* are collections of icons on a computer screen which organize input and output to/from a system. For this project, we looked at *graphical user interfaces*(GUI); where actions are done through using a pointing device connected to a computer with a keyboard for entering values and other inputs.

To create this project, we developed the communication class implementation and the *client*/*host* collaboration systems. Starting at the network socket and file I/O levels, we devised a polymorphic communicator which would ease the requirements of *client* information transfer from one to all *clients*. Once the communicator was implemented, we were able to devise the *client's* frontend to be based on a standard user interface library called FLTK which would fulfill the need for an easy to configure, lightweight dependency for operator input and output. The backend was designed to take advantage of the communication implementation to read and write data to/from the *host* over a specified network path. The *host* was designed to plug into a fellow graduate's real-time Linux data acquisition & control module when their work was to be completed. As of now, we await their work to be finished before full integration of the communication system can be done. Testing has been done to determine that it is feasible to read and write the data from the real-time module for sending and receiving data between the *host* and *clients*. Until their work reaches completion, the system will not be available for testing in a proper testing environment.

Chapter 2

Background Review

2.1 Overview

The *engine test cell*, as stated in the introduction, uses monitoring and controlling tools to gather information on an engine's operation. Monitoring tools provide procedures to produce observation data on the engine. An engine's execution is controlled to simulate and test certain conditions and events. To monitor and control the system interactively, an *interface* accompanies the *engine test cell* in a *suite*.

When designing a *monitoring* and *controlling* system, there are general *user interface suites* available. Many are actively used in the educational and commercial/industrial sectors. These *suites* allow an *operator* to operate an interface using premade graphical components on a computer. The capabilities of these *suite* are modular. Components can be added and removed without redesigning the interface from an empty initial state. *Modules* provide functions distinct and specific to their particular operation.

To enable communication between *suites* and target systems (whether local or

networked), each *suite* employs a *backend* to facilitate intra-system and inter-system communication. The different types of communication *protocols* are numerous, so focus will be on those most prominently used in mainstream *user interface suites*.

2.2 Controlling Systems

To control a system means to change the inputs, which can effect a change in the outputs of the system.

The personnel controlling the system, in contrast to those programming it, are deemed the *operators*. As the team members in charge of running the system, *operators* will use a control interface to modify the behaviour of the system. The degree to which a system can be modified depends upon the number of available states. For complex systems, an intuitive interface is desired as its main objective is to streamline the inputs available to the *operator* without hindering their ability to reach all specified states of the system.

A model of the inputs, outputs, operations, and effects can be defined through a *state machine*.

2.3 State Machines

A *finite state machine* is a model describing how a system will respond to events during its operation. We refer to these events as a finite sequence of "symbols" (inputs) into the model, which is in turn used by the model to reach an "answer" (output) (Rabin and Scott, 1959, 115). *State machines* are beneficial for understanding how a system will operate after an initial sequence of events. *States* and event *transitions*

visualize the operations of the system in a more intuitive and natural way for *developers* and *operators* than enumerated or itemized lists. (Harel, 1987, 232.p3). A simple example of a *state machine* can be seen in Figure 2.1. Each arrow originating from one state to the next is a *transition*. These *transitions* are events which conditionally mark when a state should be relinquished and operations changed to the next *state*.

The next *state* must exist in the finite non-empty set of internal *states* for the machine to be complete (Rabin and Scott, 1959, 116). Using undefined *states* removes the finite resolution of a sequence of inputs for a *state machine*. In addition to avoiding undefined *states*, a deterministic *finite state machine* must avoid non-deterministic execution.

To avoid non-deterministic execution, where the next state cannot be predicted from the previous state and transitioning event, events must satisfy mutual exclusivity. It must not be possible to arbitrarily choose between two transitions during execution (Huang, 2010, 37.p2).

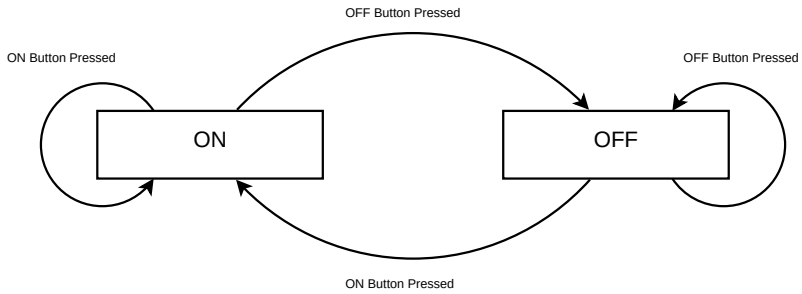

Figure 2.1: Simple State Machine

State machines can be represented in *state diagrams*, which are a type of *control graph*.

2.3.1 Control Graphs and Diagrams

Graphs and diagrams have been designed to facilitate expressiveness in representing the aforementioned systems. Through fundamental theory, Harel explains that graphs and diagrams are sets of points and nodes with interconnections, edges and arcs, for expressing some type of link between two or more entities (Harel, 1988, 514.p3). Edges and arcs join each point or node in a way relevant to the concerns being displayed by the graph. Simplifying the structure of systems into such primitive designs may cause some details to become lost or overwhelm the graphs, which is a factor in choosing certain model types. Harel states that events causing a transition from a "large number of states, such as a high-level interrupt" can result in an "unnecessary multitude of arrows" (Harel, 1988, 522).

A trade-off between node complexity (the number of details per node) and link complexity (the number of links between nodes) is one of the defining features of control graphs and diagrams. Simple graphs such as state diagrams can have high link complexities to the point where the diagram is convoluted with numerous transition links if certain events are universal to most states (Harel, 1988, 522.p4). Higher abstraction graphs such as state charts reduce the number of transition links by increasing the defining semantics required to create and interpret such graphs. The trade-off is between quantity (number of transitions) and quality (number of semantics).

A type of graph used for illustrating signal relationships is the *signal-flow diagram*.

2.3.2 Signal-Flow Diagrams

Signal-flow diagrams are graphs used to depict relations between signal outputs and signal inputs. Two important aspects of the graph are *loops* and *gain* between signals. *Gain* is the ratio between two connected nodes on the graph, which is the ratio of signal magnitude from one node to another (Edwards, 2001, 3-11.p1). *Loops* are multi-node connections where a specified node is its own ancestor and descendant. *Loops* are important for representing *feedback* relationships between signals within the diagrams. *Feedback* returns information to a signal about the net propagated effect of output. Since *feedback* is not separable into its amalgamated components, it can only be used to ascertain whether the signal caused a general increase or decrease in magnitude of the output. The only limit to the number of feedback loops in a *signal-flow diagram* is the number of unique connections between each node.

In conjunction with a *controlling system*, a *monitoring system* poses another part of the *control system* structure.

2.4 Monitoring Systems

Monitoring a system and controlling a system are different in view of their distinguished roles. *Monitoring systems* record and display the properties of the system to an external entity. Controlling a system changes the system's inputs and state to operate towards a desired output.

Parnas used his *four variable model* to relate the real world input and outputs variables to the internal software input and output variables (Parnas, 2003, 14). The four variables are the physical and virtual mappings of monitored and controlled

quantities.

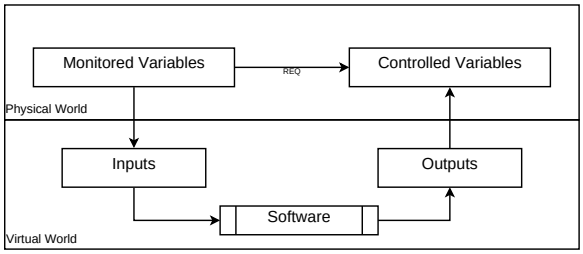

Figure 2.2: Layout of *four variable model*

Reading values into a requested form for an input or output variable may require converting the values between *continuous* and *discrete* forms. It is necessary to determine if the perceived values (values gathered by the monitor) adequately resemble the system being watched. Computer systems are inherently *discrete* in operation. In contrast to the *continuous* values, *discrete* values occur at distinct intervals in a discontinuous mode.

During monitoring, it is not possible to observe data and directly "read" it into a computer system in its original continuous form; it must be *sampled*. *Sampling* is the reading of values at predetermined intervals to get an approximation of the values when the intervals occurred.

To compare *continuous* versus *discrete* values, several major differences must be resolved. Each digital value occurs after a certain time delay inversely proportional to the operating frequency. The higher the frequency, the higher the *sampling rate*, the lesser the magnitude of time intervals and thusly a representation of the signal more closely approximating the *continuous time* signal, if the resolution of the system is the same at the higher *sampling rate* (Branicky *et al.*, 2000, 2355.p6). To increase the frequency and *sampling rate*, we need to conform to certain limitations

within components of our "monitor". Limitations may be due to physical components, connections/communications or logical properties of the system. Exceeding these limitations will result in inaccuracies or precision errors resulting in misrepresentative data.

The problem of retaining values does not stop at the relation between real world values to virtual variables, there is also a consideration about "how" these variables must be stored. To do this, an understanding of the digital representation of values is necesssary.

2.4.1 Digital Value Representation

Computer systems have the ability to store information not because of an inherent translation ability, but through executed conversions between virtual formats and binary storage. As explained previously when describing Parnas' *four variable model*, a relation between the internal binary structures of a computer and virtual values must be formed to use such values.

Floating Point Representation

Floating Point is defined by the IEEE 754 as a standard for encoding and decoding scientific numbers

Numbers encoded in this format follow a designation for the number of bits to use to represent the information required (IEEE, 2008, 9.p2):

- Sign - Whether number is positive or negative
- Exponent - Magnitude of number in base-10
- Significand - Integer value with no exponent

Integer Representation

Integers are one of the basic virtual representations of data on a computer system. Created in fixed bit-length sizes such as 32-bit, 64-bit or higher, these numbers can represent values from 2^{32} (approx. 4.29 billion at unsigned 32-bit) to 2^n for n bits (approx. 3.40 x 10^{38} at unsigned 128-bit).

One major concern when dealing with integers (especially when stored on transferrable media) is the *endian* type of the architecture the integers are being used on. *Big endian* is a storage method where the bytes are stored based on the most significant part of the computer's internal registers first. *Little endian* allocates the least significant part of the register first (James, 1990, 1.p3). When information travels between *little endian* and *big endian* systems without regards to conversions of byte order, the extracted value of the information will be incorrect. The ordering of bytes will affect the interpretation of data on the opposing system.

If all computers and devices in a working group, such as an environment or system cluster, operate on the same byte order, this point becomes moot and can be avoided.

Floating point and integer data are utilized in representing data values in low-level networked monitoring and control.

2.5 Monitoring & Control over Networking

With the continued success of the Internet and communication over inter-entity media (Ethernet, Modem, etc.), controlling systems remotely using networked computer systems has become very favourable in contrast to monolithic all-in-one systems. Monolithic all-in-one systems were systems hosting all calculations and operations

within a single computer. As requirements for such systems increased, the price increased at a higher-than-linear rate. By using multiple networked computer systems, the role of calculations and acquisition became shared. This lowered the overall cost of the system.

Communications over a network are composed of *messages*. Each *message* is an electrical transmission sent from one system to another for the purpose of exchanging information digitally. The information is encoded in a standardized fashion to allow reliable transceival on the sender and recipient sides of the network. These standards for transmitting and receiving are numerous, from industrial proprietary standards to open source free-to-use standards. Below are a few examples of physical standards for network communication (IEEE, 2013):

- Ethernet - IEEE 802.3
- Wireless Networking - IEEE 802.11
- Personal Wireless Networking - IEEE 802.15

Each standard, as stated by the IEEE, defines the specifications for implementing compatible communication devices. For general purpose communications, the open standards work well enough in addition to being under lesser restrictions than industrial proprietary standards. One major advantage of using these open standards is the overwhelming number of off-the-shelf devices ready for purchase from manufacturers. Depending on the properties required, these devices can perform the lower-level physical communications without having to put a larger load on the computer systems executing the *backend*, if such communications are necessary.

These devices will take care of forwarding and receiving low-level *messages*. One

of the major issues with communicating between systems over a network is the importance of timing between *messages*. If a *message* is sent to another system, it must be received within a pre-defined amount of time. If the *message* takes any longer than expected, it will *timeout*. *Timeout* is the failure in which the *message* is determined to have been lost or never sent. Even if the *message* reaches its destination after being timed-out, it will be discarded as an error since the recipient wasn't expecting it anymore, as seen in Figure 2.3. Without being able to *timeout messages*, communications would never be able to shut down, as it could miss a *message* still on its way.

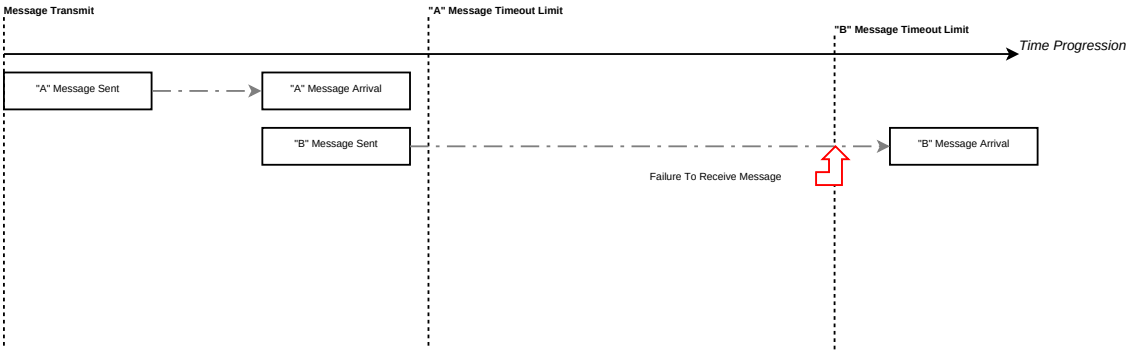

Figure 2.3: *Message* lost due to *timeout*

The maximum length of time between *messages* has a different effect; the lower the maximum time between them, the higher the number of *messages* which can be sent per second. This leads to a higher data rate. Any events, such as overburdening the network with enough traffic to overwhelm the transmission medium, will cause the timing between *messages* to increase. This will lead to a drop in the effective data rate (Branicky *et al.*, 2000, 2355.p6).

Being able to send a *message* is only one part of the problem. To sufficiently trust the data sent as being relevant for the current time frame, we need to be able to know

that the *message* has arrived without significant delay. If the delay is too long, the information sent may not be correct for the time being. This is where *deterministic* networking is necessary. Not only must a *message* arrive, it must arrive within a specified time frame (Galloway and Hancke, 2012, 3.p2).

2.5.1 Deterministic Networked Control

In networked control, the determinism is related to the reduction of "randomized" properties of the system. If *messages* arrive haphazardly with no precision to when they arrive or are sent, the system cannot be *deterministic*. For *deterministic* networked control, we need a system with strict limits on the maximum length of time between *messages*. Being able to limit the length of time between *messages* to a finite range is not the only problem inherent in deterministic networked control. For the receiving system to operate on the data it reads from the transmitting system, the information must be "fresh". The data must be relevant to the current value. If the system is receiving data after it has changed, the information loses its relevancy and becomes "stale".

As stated by Halpern, the information available to each entity will limit the actions they can accomplish successfully (Halpern and Moses, 1990, 1.p4). To know what this "information" entails, we need to give an adequate description of what it is and why it is important. If we imagine the delay between systems as a barrier between local and remote "knowledge" (information), or location-based data, we can infer that a system cannot work unless it can resolve some distant remote data into local data for further operations (Halpern and Moses, 1990, 7.p1). Even after resolving such information, it is to be assumed that the information could already be expired and changed. We can

never perfectly synchronize local and remote information with non-negligible delay, so we must make some trade-offs.

Since we are working with a discrete system, we can assume that by following what was said by Galloway above, by keeping the network transit time to be shorter than the time between data changes, we can assume the information on both sides of the interconnect to be "synchronized" (Galloway and Hancke, 2012, 3.p1). This skips over the fact that networks are not absolutely reliable in their transmissions; any messages lost, even after being retransmitted, will affect the synchronicity of the systems if the entire process takes longer than the time taken between data changing.

While controlling a single system with a single controller is relatively straightforward and direct, having multiple systems being able to connect simultaneously to a single *host* is a different matter. In a distributed form, where each system has some non-negligible delay between each other, we cannot assume that information will reach each *client* at the same time. The delay, however small, will affect the outcome of systems sensitive to the timing of receiving information.

To better understand the effect of order and timing of events, we can observe the properties of *causality*.

2.6 Impact of Causality

Causality is a diverse topic surrounding the cause-and-effect of serial and parallel events occurring in regards to one another. *Causality* is a major concern in the case where events affect the progression of one another. Given event "A" and "B", if "B" cannot exist until "A", it is said "B" depends on "A" (Lamport, 1978, 559.p7).

As discussed above in the distributed networking section, we need to understand

consistency between transmitters and receivers. The messages sent are assumed to be transmitted successfully and, if the protocol allows for it, are organized in successive order when received. Transmission control protocol (TCP) is such a protocol, which helps alleviate order-of-receipt problems, but it is not the only applicable protocol for networking. Other protocols, such as user datagram protocol (UDP), can be used with or without adaptation if it is constructed in a way to avoid causal conflicts and problems.

The focus on causality with respect to computer science will be on the analysis of *deadlocks* and *race conditions*.

2.6.1 Deadlocks, Race Conditions & Causality

For systems, regardless of their digital or analog mechanisms, we must be aware of the way these mechanisms interact with each other to avoid non-deterministic outcomes.

In computer science, a very well known yet hard to predict problem is the existence of *deadlock* and *race conditions*. *Deadlock* is an event where two synchronous event progressions reach a point where both require the other to finish its current state before the waiting event can continue. Once a system reaches this state, an external method must occur before they may exit. These methods consist of hard resets, interrupts, or other such interjections of control.

Race conditions are symptoms of synchronous and asynchronous event progressions where the time at which both events reach and finish, a specific state is ambiguous. The system will react differently in relation to whichever event is resolved as finishing "first" (Schwarz and Mattern, 1994, 1.p1). Without mechanisms such as semaphores, mutexes, memory/progression barriers, and signals to avoid such a

problem, both *race conditions* and *deadlocks* can occur frequently and/or sporadically.

Even if a system reaches *deadlock*, it is hard to detect this without some method of reading or inferring the overall state of the system. With a system of multiple components working together, the status may only be found by resolving states into "explicit knowledge" (Halpern and Moses, 1990, 7.p1).

One of the current paradigms to organize and operate components in software is *object-oriented programming*.

2.7 Object-Oriented Programming

Object-oriented programming is one of many paradigms used for organizing the flow and expansion of concepts and ideas in software systems. As a deep and expansive paradigm, only a small subset of features will be described below for their relation to the software project.

Object-oriented programming has methods of encapsulating procedures and data into a hierarchy of *classes*. *Classes* define inheritable procedures and storage structures which can be directly expanded upon through *inheritance* (Cook, 1989, 1.p2). Implementing programs using *object-oriented programming* can be done through separating concerns into distinct components. To do so efficiently and effectively requires fully analyzing the system design for interdependent and independent components.

Inheritance is a design choice where ancestral, lower level objects share properties and methods to descendant, higher level objects. The reasoning for *inheritance* is to reduce duplication of properties and methods which exist in multiple locations and perform the same tasks. By organizing the definition of structures to reside in shared locations, design changes can be done without increasing the required work to

implement these changes in an existing infrastructure.

Classes are a collection of properties to express an *object's* "blueprint". Each *class* dictates the properties and methods of an *object*, and from where they receive these methods and properties from. *Classes* are beneficial in localizing methods and properties which are important for the defined *objects* to execute correctly and efficiently.

A major step towards effective *object-oriented programming* is high object *cohesion* with low *coupling*. *Cohesion* in programming is how related the functionality of higher-level *object* are to lower-level modules. High *cohesion* means unrelated modules are not closely grouped with other modules. The aim of high *cohesion* is to keep *objects* organized and efficient, by only containing the components they require. *Coupling* in programming outlines how interconnected different *objects* are to one another. In an ideal object-oriented program, *objects* are connected to one another only when required to complete their tasks. Creating dependencies between *objects* only for some of their modules functionality means the module should be moved up to a higher level and "inherited" to achieve the same means with lower *coupling*.

Object-oriented programming can be found in many programming avenues. One such avenue are *user interface design*.

2.8 User Interface

To view, interact, and control systems, we need a *frontend* with which to communicate an *operator's* intentions. Some interactive elements may be as simplistic as buttons, dials, and lights on a panel, while others have been designed to be visible and emulated on a computer screen for abstracting away the complexities of the underlying system.

For computer systems developed over the last few decades, the visual system has become increasingly complex.

As explained by Wills, the method of communicating data from the system to the *operator* is a type of "dialogue". These interactions can range from command-line interfaces where typing is the main input method, to highly graphical displays containing collections of images and text (Wills, 1994, 417.p7). Current consumer computer operating systems use these user interfaces explicitly for simplifying all actions down to the use of a movable on-screen pointing image and graphical entities.

For many consumers, the user interface is the focal point of their interactions, as it is the part people can directly control and view; from being able to watch its graphical outputs to the movement of on-screen elements to send inputs to the system (Galitz, 2002, 4.p2). An interesting aspect of isolating all elements onto a screen is the effect of graphical positions of inputs and outputs to a user; increasing the confusion and distortion of elements increases the time required to operate the system for similar work flows through extending the amount of time required to verify corrections of inputs and outputs (Galitz, 2002, 5.p3).

While building user interfaces from the ground up is possible, *user interface suites* have made it possible to generate adequate interfaces with very little effort required.

2.9 User Interface Suite

User interface suites were designed to produce a common set of tools for programmers to create interfaces. By creating a standard set of tools, making interfaces should be easier than if the interface had to be designed from the bare operating system basics (Galitz, 2002, 23.p8).

Some *suites* are "multi-platform", which makes it easy to move a project between different operating systems. For a *suite* to be "multi-platform", it must be able to build on multiple operating systems with no platform-specific modification required.

The images and icons on a *user interface* are usually linked to data storage in the back-end of the *user interface suite*, which allows communicating information to and from the backend (Xudong and Jiancheng, 2007, 540.p9).

This project's focal type of *user interface suite* is Signal-Flow UI suites. These types of *suites* use the step-by-step evolving state of the model to generate the output data rather than symbolically solving the system before extracting the final data.

2.10 Signal-Flow User Interface Suite

Signal-Flow user interface suites, model systems by simulating the flow of information as *signals*. *Signals* are routed between virtual objects during time intervals. As each signal passes through an object, it is altered to reflect new information attained from each particular object in its path. By modelling the progression of the system from one time frame to the next, the *signal-flow user interface suite's* backend can model systems which cannot be easily broken down into equations and formulae.

The visual elements of a *user interface suite* have become synonymous with "drop-in" design components. These elements make up what can be classified as "drag-and-drop graphical user interfaces".

2.10.1 Drag-and-Drop Graphical UI

Most *suites* that fit under the designation *signal-flow user interface suite* use draggable elements on-screen to arrange and output information. These "draggable" elements can be moved through the use of an on-screen pointer to be positioned in relation to the application as necessary.

When designing an interface for use, it is imperative to keep the arrangement of symbols and elements as compatible with the *operators* whom will be using the interface as possible. Of course, for designing the software which will be used by *operators* to create derivative interfaces, it is favourable to implement certain "features" to limit the discord which can be created by disorganized *operators* (Galitz, 2002, 24.p7).

Mathworks has a specific *suite* under the name MATLAB, with a feature called Simulink.

2.10.2 MATLAB(Simulink)

MATLAB's Simulink is a product built into Mathwork's MATLAB *suite* to perform data processing and simulation with both physical and virtual data. Simulink has the tools required to create a user interface with blocks to facilitate signal flow processing and manipulation. The blocks shown in the user interface *suite* appear to be based strongly on the block diagrams explained beforehand. As stated by Rajagopalan, the information generated by blocks is sent to other blocks internally, depending on the graph "lines" connecting one block's output to another (Rajagopalan and Washington, 2002, 4.p2).

The inherent values generated by blocks in Simulink are causal in nature; values cannot be created from information in the future unless those values have already

occurred and the workflow is being executed as if it were simulating the future. For causal, non-linear systems, Simulink can numerically solve the system, avoiding some disadvantages to solving the systems analytically. The nonlinearities are simplified and calculated during solving steps, or "iterations", where each time sample can be used as a snapshot of how the system has evolved to this point (Rajagopalan and Washington, 2002, 12.p1).

Using the library of ODE solvers, Simulink has the ability to solve certain classes of differential algebraic equations (DAE). One example is the use of the "ode15i" solver to solve differential algebraic equations of index 1 (Mathworks, 2013). Other equation-based solvers have the ability to solve higher index cases symbolically instead of numerically. Modelica provides an example of problematic higher index DAE where it must be reduced symbolically before numerical solvers can be applied to generate an accurate and reliable answer (Association, 2012, 253.p8-9). High index DAE solutions in general cases are not available currently in 2013 for Simulink.

National Instruments LabVIEW is another similar example of a *suite* providing numerical analysis and signal-flow user interface construction.

2.10.3 LabVIEW

LabVIEW is a data processing suite created by National Instruments Inc. It includes an array of modules and libraries for creating simulators and interfaces to monitor, model, and control systems. Modules in LabVIEW are components used with one another to create a virtual system. *Operators* can link into a real world system and/or simulate a similar system using these components. LabVIEW has some features which are used to help produce user interfaces and simulators that work concurrently with

each other. Some features, like the Event Structure, are used to expand on the paradigms available for the engineer or operator creating a respective project (Smith, 2012, 1.p3).

Each LabVIEW project is called a "Virtual Instrument". In a "Virtual Instrument", modules are represented by widgets which are organized on the "Front Panel". These widgets have an onscreen appearance which differentiates one type of widget from another. In the case of LabVIEW, each widget represents a corresponding input/output block which appears on the "Block Diagram" screen. The outputs of these blocks are linked to the inputs of one or more blocks to form a flow of signals. While the project is executing, the inputs and outputs will be updated to reflect the state of the system and any attached devices. Devices can be data acquisition devices, or any other supported peripherals.

A "Virtual Instrument" can be changed only if it is not currently executing. If modules are to be added, removed, or changed, the project must be stopped. Once the changes are complete, the project can be executed again.

The modules available to LabVIEW are proprietary. To use such modules, a license must be obtained from National Instruments Inc. The suite is closed source which does not allow viewing and modification of the core program source code. In addition, the majority of drivers used to connect to peripheral devices are Windows-only. For any projects which are to be run on other operating systems, this will hinder development.

Chapter 3

Requirements

3.1 Introduction

3.1.1 Purpose

The purpose of this chapter is to define requirements of the *engine test cell suite*. Requirements specify what the *engine test cell suite* needs to fulfill to accomplish its role in diagnostics. We define two major types of requirements and expand on the requirements needed to create a working *engine test cell suite*. Decisions with regards to how the *engine test cell suite* will operate are discussed in section 4, the design chapter.

3.1.2 Conventions

Interactive software has a *user interface* which is backed by a layer such as the "business logic" or "processing layer". When referring to the *engine test cell suite* "user interface", it is called the *frontend*. When referring to the *engine test cell suite's*

"business logic", it is called the *backend*. When referring to the "data acquisition and control unit" attached to the engine as part of the *engine test cell suite*, it is abbreviated as DAQC. The users of the system are the *operators* and *developers*.

The *operators* are the standard users of the *frontend*. Their purpose is to operate the system in day-to-day *engine test cell suite* diagnostics for testing and reviewing engines. *Developers* are users whose role is to program the *frontend* and *backend* of the *engine test cell suite* to operate in a specified fashion.

3.2 Overview

The *frontend* and *backend* are described in terms of their requirements from *operators* whom are expected to be the primary decision-makers in control of the software. The requirements are split into two major groups, *safety requirements* and *operating requirements*.

3.2.1 Operating Requirements

Each of the requirements described under *operating requirements* outline requirements which keep the system working as specified. While failing to adhere to the *safety requirements* is critical to the safety of the operator and surrounding environment, failing to adhere to these *operating requirements* will affect the efficiency, reliability and ease of use of the system.

3.2.2 Safety Requirements

Each of the requirements described under *safety requirements* are employed as risk mitigation properties of the system. These requirements help to avoid critical situations which may arise during operation if proper techniques are not upheld.

Both the *frontend* and *backend* share the following safety requirements:

1. Availability (Controls must not block other controls)
2. Safe (*Operator* and plant must be safe from damage during operation.)
3. Reliable (If operating environment changes {network traffic, latency, etc.}, the system must continue operating as specified.)

If the system becomes unavailable, the *operator* risks losing control of the system when problems arise. If the system becomes unsafe, the aforementioned loss of life or system integrity is more likely to occur. If the system becomes unreliable, errors can cause failure to operate as designed. Watching the system to see whether it is still working correctly can be done through a watch dog or through a feedback system to keep track of unresponsive components. This will be discussed in the *design* chapter of this thesis.

3.3 Frontend Requirements

The *frontend* of the system is the *user interface*. *Operators* of the system will interact with the *engine test cell suite* through widgets in this interface.

3.3.1 Operating Requirements

As a *user interface*, the *frontend* contains widgets. Widgets are objects with an on-screen appearance. The widgets fit into the following categories:

1. Calculators
2. *Operator* input
3. *Operator* output

Calculator (scientific) widgets calculate scientific features from data. *Operator* input widgets take information from the *operator*, which is then be sent to other widgets. *Operator* output widgets take information from other widgets and present them to the *operator*.

3.3.2 Widget Requirements

Of the categories listed in the *operating requirements* section, there are specific types of widgets which are widespread and well used on interfaces. These widget types are listed below, along with which category they fit under.

In Figure 3.1, the intersecting boxes illustrate combinations of multiple categories. Some utilize communicating information across to other machines or entities in addition to outputting information to the interface, while some may use both input and output from the *operator*. Information, such as channel values and incoming data, are sent to the *frontend* by the *backend* to be outputted by output widgets. Widget input data is sent to the *backend* by the *frontend* to be used as specified.

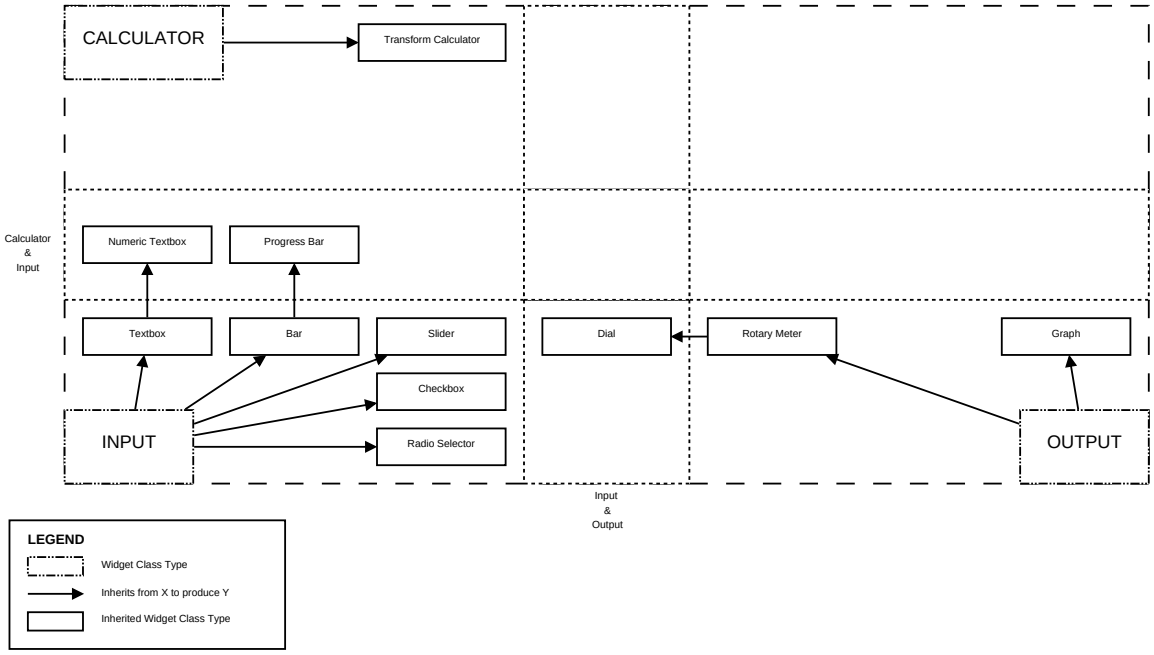

Figure 3.1: Widget types represented by operation type and example inherited types

3.4 Backend Requirements

The *backend* is the business logic of the *engine test cell suite*. The purpose of the *backend* is to execute procedures initiated by the *operators* while receiving data from the outputs of the *engine test cell suite* to be displayed on the *frontend*.

3.4.1 Operating Requirements

The requirements described below dictate what the *backend* must adhere to for optimal operation of the *engine test cell suite*.

1. *Backend* must convey main *operator's* change requests to the DAQC.
2. *Backend* must convey DAQC data to the *frontend* for all *operators*.

3.4.2 Communication Requirements

The communication component of the *backend* consists of procedures in transferring and receiving information between the *backend* and the *host* of the *engine test cell suite*. The mutual relationship between the *frontend* and *backend* is the following:

- The *frontend* requires communication from the *backend* for receiving data from the DAQC.
- The *backend* receives data from the *frontend*, which is transferred to the DAQC.

For the communication medium chosen by the *backend*, there should be no impact on the *frontend*. The *frontend* is only required to communicate in a specified manner to the *backend* about changes initiated by the *operator*.

3.4.3 Host Requirements

As a part of the system hosting the DAQC business logic and being the common end point for *client* communication channels, the *host* is required to adhere to the following:

- The *host* must apply requests received by the main operating *client* only if active and in safe operating conditions.
- The *host* must transmit all updates received from the DAQC to the connected *clients*.
- The *host* must enter emergency shutdown upon request from main operating *client*.
- The *host* must restart operation only after request from main operating *client*.

The main operating *client* designation will be described further in the design section of this thesis. For the purposes of the requirements section, it is described as

the only *client* with the ability to modify the DAQC outputs during operation of the *engine test cell suite*.

Chapter 4

Design

4.1 Frontend Design

The *frontend* of the system is the *user interface* used by the *operator* to trigger changes in the *engine test cell suite* through the *backend*. The *frontend* uses widgets to form the *user interface*. In addition, the *frontend* receives data describing the status of the *engine test cell suite* via the *backend*. As described in the requirements section of the thesis, we will outline the role of the main operating *client* with regards to the other monitoring *clients*.

4.1.1 Operating Design

As stated in the requirements chapter, the *frontend* contains widgets. For these onscreen objects, each widget must conform to the following specifications:

1. Widget must specify the type of input it accepts.
2. Widget must be concise and functionally isolated.

For every input widget, the input type must be specified. If a widget accepts only numbers, then a non-numerical input must not be accepted to be transferred to the *backend*. With functional isolation, widgets must not have side effects which alter the operation of other widgets. The operation of a widget is constant until the target procedure is changed to a different operation.

4.1.2 Safety Design

Design choices which affect the safe operation of the system are noted below:

1. Widget must not block other widgets.
2. Widget must be non-modal.
3. Widget must display the value passed to them as-is.

If a widget blocks another widget, the *operator* cannot operate the interface in all online conditions. Online conditions are conditions where the *engine test cell suite* is executing a test. In addition to avoiding widget blocking, the widgets must also be non-modal. Changes to one widget must not disable access to other widgets.

Data from the *backend* must be kept intact before showing it to the *operator*. If the *frontend* modifies the data from the *backend*, it falsely represents the data as specified prior to modification. The data being displayed to the *operator* in this case is not consistent with the data being stored in the *backend*. Falsely representing data may show faults where none exist (false positive) or hide faults where some exist (false negative).

If a widget fails to send critical control data to the *backend*, such as control stops or restrictions, the *engine test cell suite* becomes dangerous to the *operator*, fellow workers, or operating environment.

4.1.3 Interface Modes

The *frontend* has three modes, "Initialize Interface", "Create Interface" and "Interface Online". Each mode focuses on one of the following tasks:

- Linking the *frontend* with the *backend*.
- Creating, loading or saving the interface of the *frontend*.
- Using the *frontend* with the *backend*.

Initialize Interface

"Initialize Interface" mode attaches the *frontend* to the *backend*. Once connected, the *frontend* will switch to the proper interface setup and continue to "Interface Online" mode to supervise and operate the *engine test cell suite*. Error checking must be done to ensure the system retains reliability for all upcoming operations.

Create Interface

Operating in "Create Interface" mode allows the *operator* to move and add widgets to the *frontend*. As long as this mode is enabled, each widget will be adjustable, removable and importable. Any changes will be saved and loaded for future use. This is done with modifying the interface to facilitate new *frontend* requirements.

Interface Online

Operating in "Interface Online" mode allows the *operator* to interact with the widgets on the *frontend* without the ability to modify where the widgets are, and what effect the widgets cause. This will be done to keep the interface uniform during operation. During the "Interface Online" mode, the *frontend* will send all value changes while

operating as the main operating *client* to the *backend* while accepting value changes from the *backend*.

4.2 Backend Design

The *backend* works to keep input values bound to safe limits during operation. The *backend* also synchronizes input and output data with the *host*. The data received from the *host* is processed and further sent to the *frontend*.

4.2.1 Operating Design

When designing the *backend*, there are several key aspects which must be planned for before constructing the system. The operating *designs* are listed below:

1. *Backend* must provide *frontend* procedures to send and receive data to/from the *host*.
2. *Backend* must provide error-checking within procedures to avoid generating malformed data.

4.2.2 Safety Design

Safety designs are designs related to the physical safety of the surrounding environment and *operator*. As such, the following *designs* outline features which help increase the safety of the system.

1. *Backend* must have an emergency stop procedure to send an emergency stop request to the *host*.

2. *Backend* must be reliable in sending and receiving information to/from the *frontend* and *host*.

The emergency stop procedure returns the outputs under control by the *host* to their default initial values. As this does not take into consideration the physical limitations of the devices attached to specific outputs, shunts may be required to protect the physical devices from the change in output to the default value. Shunts must be used if required by manufacturer specifications on specified inputs.

The *backend* needs to enforce safety when operating the *host*. Being reliable is done through keeping the system operating as specified. To do so, message integrity, message transmission and message receipt need to be monitored for errors. This is outlined in the following section 4.2.3.

4.2.3 Network

The specifications described below describe how the *backend* interacts with the *host*.

1. *Backend* connects over Ethernet to the *host*.
2. *Backend* must resend lost data packets.
3. *Backend* must discard malformed packets.
4. *Backend* must reconnect to *host* in case of disconnection.

The *backend* works as a synchronizer with the *host* over the network. Any changes done by the main operating *client* connected to the *host* must be synchronized with all other *clients* to ensure consistency of *operator* observations.

4.3 Host Design

The *host* attaches to the DAQC for the duration of use of the suite. During operation of the *engine test cell suite*, the role of the *host* is to modify the set points of the *engine test cell suite* through updating the inputs of the DAQC while reading incoming data from the DAQC. These values are transmitted to all connected *backends*.

4.3.1 Safety Design

The safety concerns of the *host* focus on the interactions with the DAQC and the effects of the DAQC's outputs.

1. *Host* must keep DAQC from exceeding the power and voltage limits of the engine dynamometer as specified by the dynamometer manufacturer.
2. *Host* must keep DAQC from ambiguous unpowered outputs.

To keep the system from becoming damaged, the *host* must avoid setting the DAQC outputs to values which exceed the physical limits of the *engine test cell suite* dynamometer. Increasing outputs past physical limits and/or leaving outputs undefined can lead to safety issues endangering the *engine test cell suite* and *operator*. Values must have the ability to be bound to ranges which satisfy the specified safety concerns of the *engine test cell suite*. To know that the system is fully bound, we must inspect each input/output relation in the system with regards to the outputs controlling the dynamometer and other such actuators. For each output connected to a physical system, the specified output must be limited to a range not exceeding the manufacturer specified limits for the actuator input. If part of the system is a designated black box in terms of its operation, any outputs connected from such a system must have a limiter placed between the black box system and the actuator to

enforce the limitation.

4.3.2 Operating Design

1. *Host* receives connections over Ethernet from *backends*.
2. *Host* transmits DAQC outputs to all connected *backends*.
3. *Host* must discard malformed packets.
4. *Host* keeps track of active *backends*.

During operation, the *host* is responsible for ensuring synchronization data from each *backend* is propagated to all other active *backends*. Each *backend* works in tandem with the *host* to ensure the DAQC is outputting the values requested by the *operators*, and that the *backends* receive the data specifying the current input values of the DAQC.

During design, it was assumed that the *host* would host business logic to assert unsafe combinations of controls by defining a specific *client* as authoritative. This *client* has the authority to send shutdown requests to the *host*, which will bring the *host* into an emergency shutdown state. All other *clients* may send shutdown requests, which will be forwarded by the *host* to the main operating *client* to act upon. In terms of updating values on the *host*, only the main operating *client* may send these change requests. All *clients* will receive the updated values as specified in the beginning of this subsection.

4.4 Design Focus

The project, as stated in sections 1 and 3.1.2, is an *engine test cell suite*. The *frontend* and *backend* suite synchronizes data over a local area network. When fully implemented, it will be possible to view the output data from the *engine test cell suite* on each *frontend* with the ability to connect more *backends* to the system during execution. In addition to viewing the data, it will also be possible to modify the set points on the *engine test cell suite* using the main operating *client*.

4.4.1 Widgets

Adding and removing widgets from the screen is done during configuration. Configuration is done either *online* or *offline*. If the changes are allowed to occur while the interface is executing a test, the changes are *online*. If the *user interface* must be shut down before changes can be done, the changes are *offline*.

Offline Configuration

As an example of *offline* configuration, LabVIEW and Simulink must be changed while the system is in *offline* mode. Once the *user interface* begins execution, repositioning elements on-screen cannot be done. The advantage of this type of interface is the simplicity of state. The system is executing or it is being organized and set up for execution. Mixed cases where the system is being reconfigured while important events occurred are not possible as long as configuration and execution were mutually exclusive. A disadvantage of this type of interface is the need for down-time. One cannot expect the system to be permanently executing while allowing changes as any changes to the layout require shutting down the interface for modification. An

interesting case occurs with having multiple interfaces connected to the same *host* system. If an *operator* wants to keep track of the system without losing incoming information, rolling reconfigurations of each interface need to be designed into the system. Each executing *user interface* is sequentially shut down, reconfigured, and restarted. At least one interface is actively executing while the *engine test cell suite* is running. With proper organization, no major events are lost. The additional effort required for synchronization would be on the *operators* running the *engine test cell suite*.

Online Configuration

For *online* changes to the system, there is no need for down-time when reconfiguring the screen. Changes can be done while objects are still rendering data to the *operator*.

One advantage of this system is less down-time. Removing down-time means systems can be designed to operate even when they must be altered to conform to new configuration changes. On-the-fly changes create possibilities for quick and dynamic testing environments, which can respond to observations of missing data opportunities and other events.

A disadvantage to this type of interface is the problem of separation of concerns. When operating with *offline* configuration, the distinction between "configuration" and "operation" is done by whether the system is currently executing or not. With *online* configuration, there is no distinction; changes can be done during any point in the *engine test cell suite's* operation. Without proper design limitations, unauthorized *operators* may alter the system accidentally, negatively affecting the outcome of testing or inadvertently causing loss of control within the system. This is one of the

flaws of removing mutual exclusivity between "configuration" and "operation". After deliberation, we decided to use *offline* configuration, to avoid the situation where two differing systems instantiate a conflicting change in the configuration of the interface.

4.4.2 Reliability

For successful data transfer within the *engine test cell suite*, data reliability is crucial. Any data missing after transmission can lead to incorrect operation of the *engine test cell suite*. Missing control data will result in missing steps outputted by the DAQC system.

A desired feature of the *backend* is "continued reconnection" for network communications. If the network connection does not stay consistently connected to the *host*, the *backend* will recreate the connection. This is done until either the connection is re-established or the *engine test cell suite* is terminated. Information which fails to be sent will be queued up to be sent again once the connection is properly re-established.

With respect to the two major protocols, UDP and TCP, the reconnection system will differ for each. The UDP connection will use a wake-up packet to keep a virtual connection established between the *backend* and *host*. Each of the aforementioned features described help bring a connection back to working status without interjection by the *operator*. Even with these steps, packets lost due to network transmission and reception problems are still an issue, but the problem of manual reconnection will not be such an issue.

The communication subsystem of the *backend* is required to be protocol compatible with the communication subsystem of the *host*. "Protocol compatible" means messages comply with the protocol through selectivity of information contained within

the message. Any messages which appear protocol-incompatible will be discarded, as accepting corruption will alter the state of the *backend* and *host* indeterministically. This is due to the nature of corruption as a random alteration of the data through environmental factors such as network traffic and transfer medium imperfections.

In terms of security, the system trusts each client to not be malicious. Inputs are assumed to be set by *operators* trained in the proper operation of the interface as to not impede on the safe operation of the *engine test cell*. Security considerations are listed in the future considerations section 7.2.

4.5 Frontend & Backend Components

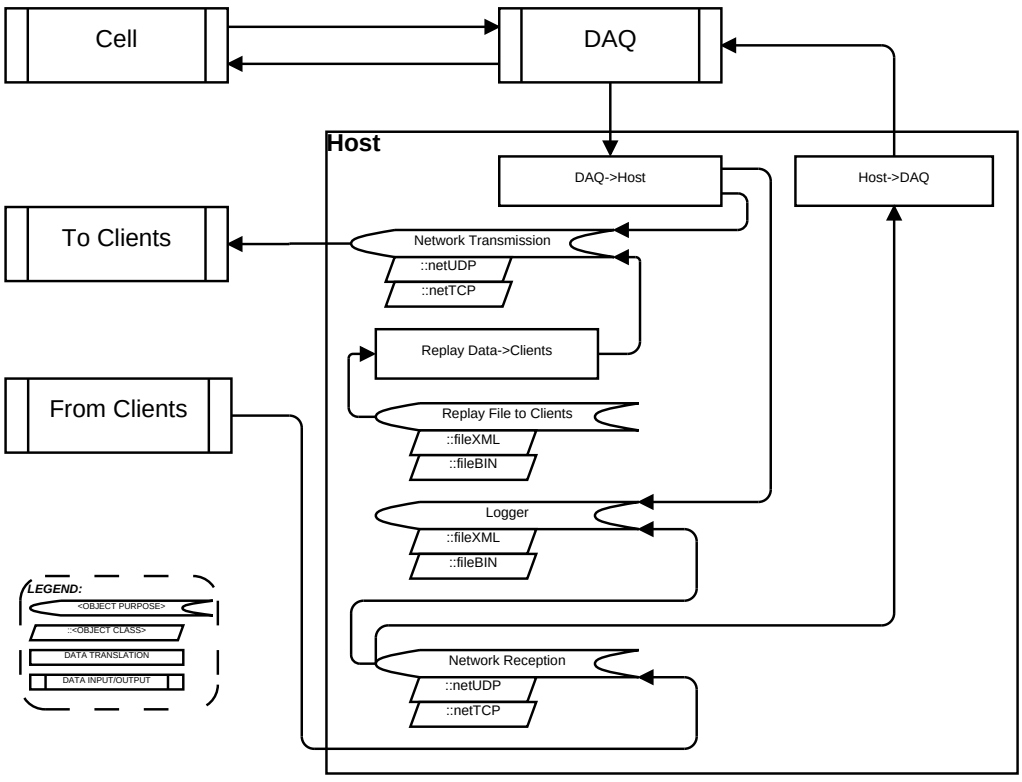

Figure 4.1: Flow of data throughout *engine test cell suite host*

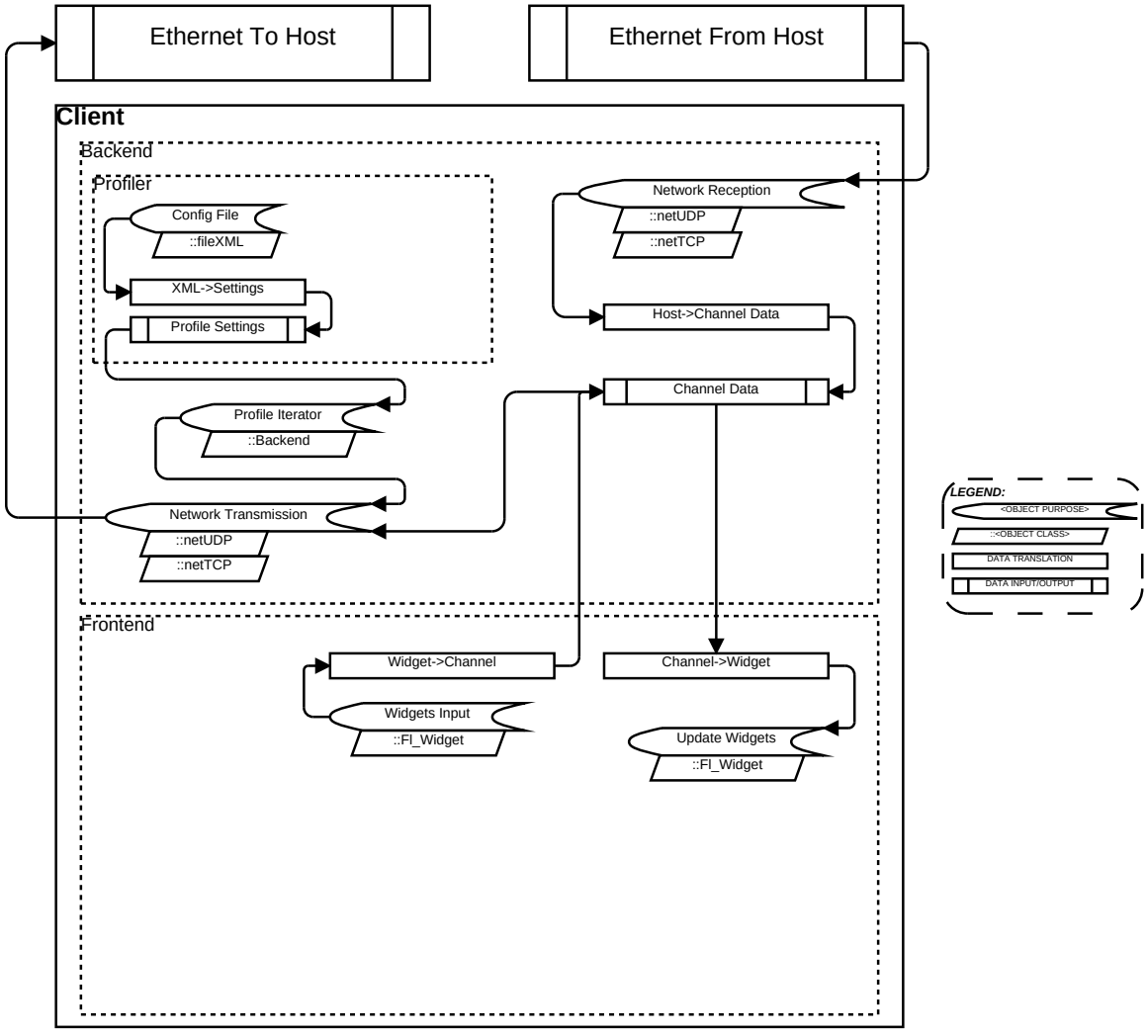

Figure 4.2: Flow of data throughout *engine test cell suite clients*

The information within the *engine test cell suite* travels from the DAQC through the *host* to the *backend* and *frontend* by a series of components composing the *engine test cell suite*.

4.5.1 Widget Management

Frontend operations occur through interacting with input widgets. Widgets are mapped within the *frontend* to channels. Value changes are sent to the *backend* during widget update *callbacks*. *Callbacks* are procedures called by event handlers when events such as *operator* interactions and system interrupts occur. When the *operator* activates a widget through input of data, its linked *callback* proceeds to send the data to the *backend*.

If the sending *client* is not the main operating *client* for the *engine test cell suite*, the updates will not be accepted by the *host*, and the *backend* of the offending *client* will receive an error message stating that such an operation is not permitted.

In the case of output widgets, data received by the *backend* from the *host* is sent to the *frontend*. The *frontend* synchronizes all widgets awaiting updates for the specified channel of the *engine test cell suite*. For output widgets with a history of values, such as a chart as shown in figure 4.5, the data is appended to the previous data to show a progression of values.

4.6 Visual Representation

The collection in figure 4.5 shows a selection of widgets available from the FLTK library. Widgets appear in a form determined by their instantiated features. *Widget*

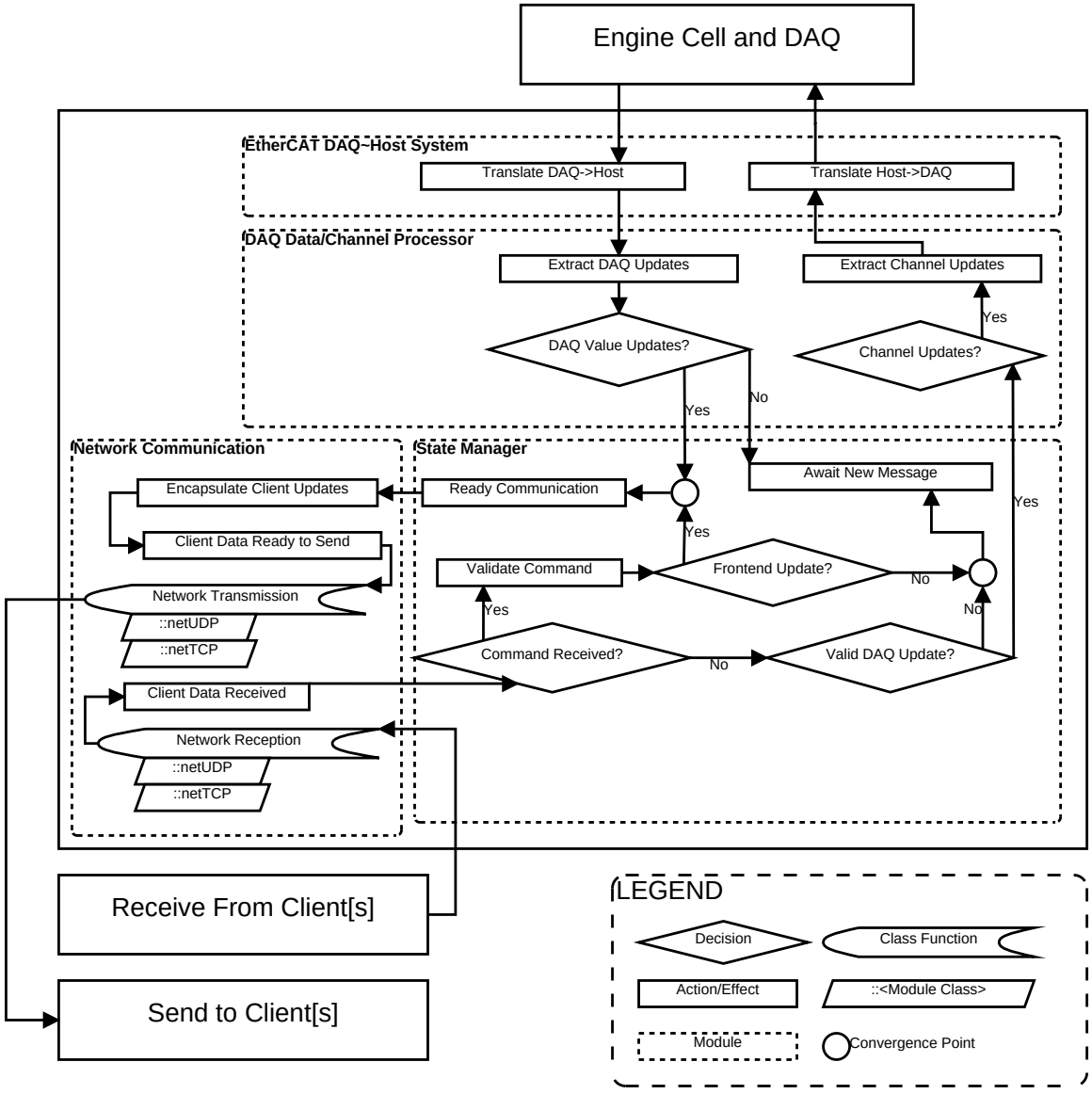

Figure 4.3: Control flow throughout *engine test cell suite host*

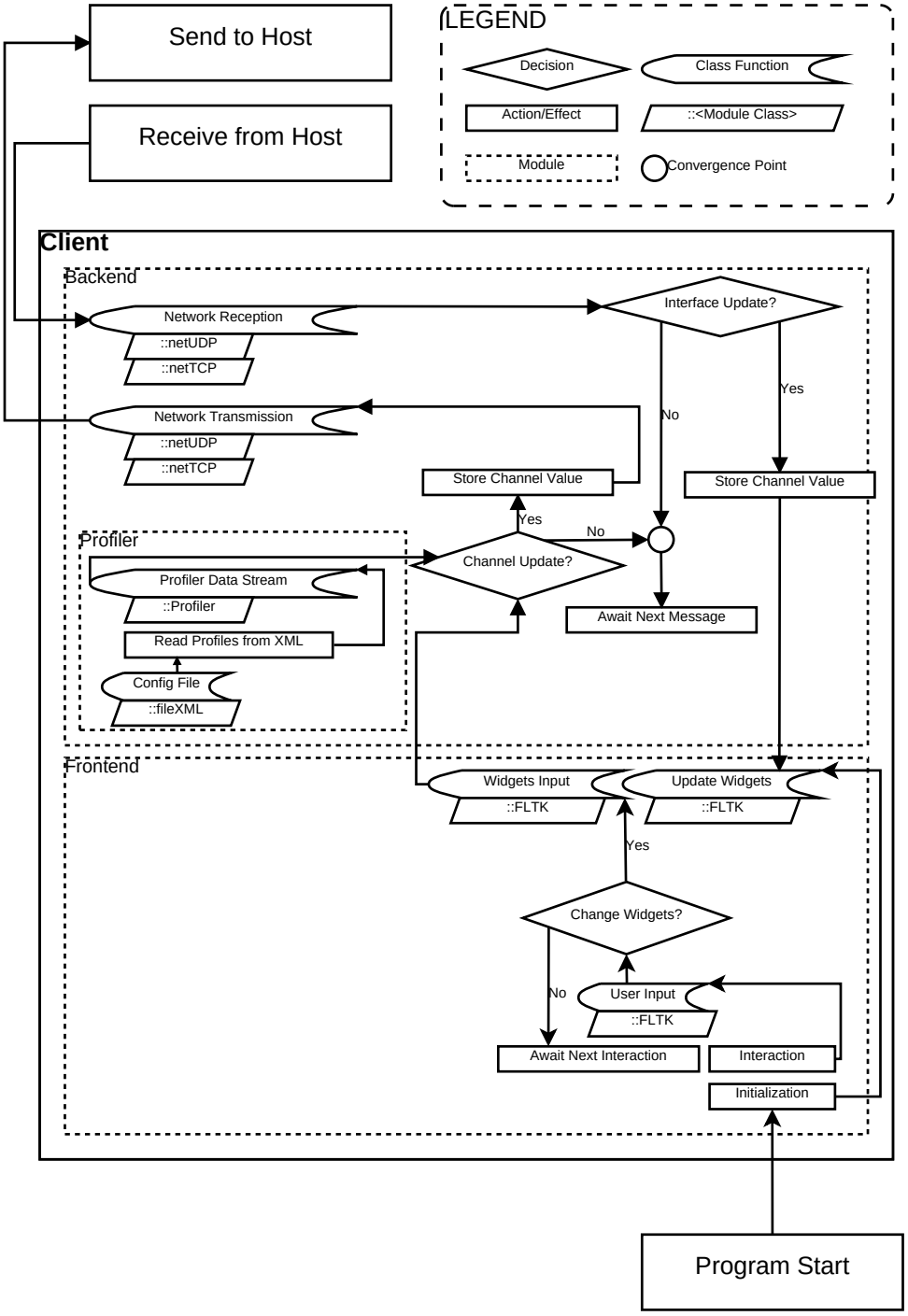

Figure 4.4: Control flow throughout *engine test cell suite clients*

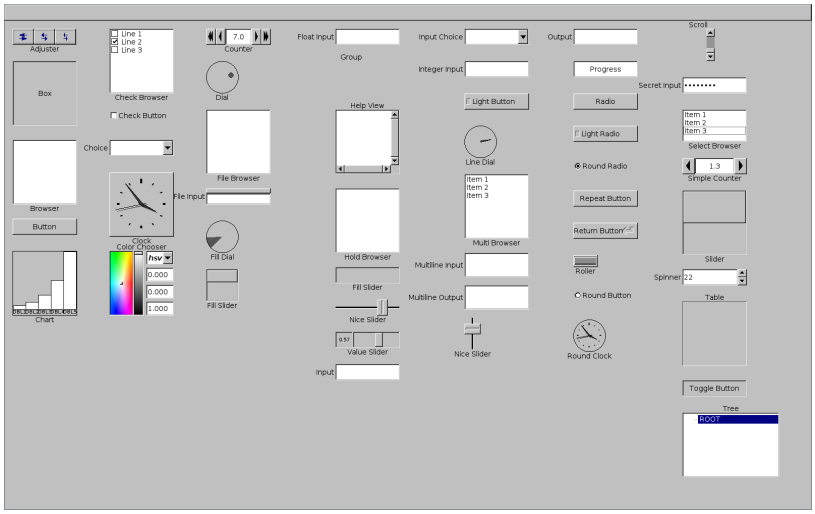

Figure 4.5: Selection of FLTK widgets for displaying and interacting with the *frontend*

features define how a specified widget will appear on the *frontend*.

4.6.1 Widget Features

Widget features specify the appearance of widgets on the *frontend*. Some of the features are:

- Width of widget
- Height of widget
- Horizontal position of widget
- Vertical position of widget
- Widget name
- Widget colour[s]

When a widget is displayed on the user interface, its features are used to create the visual representation of the widget on the *frontend*. The combined appearance of each widget on the *frontend* compose the interaction view of the system.

Widgets which are interacted with by the *operator* use *callbacks* to enable widget control and feedback.

4.6.2 Callbacks

Callbacks are a tool for executing procedures when changes occur to a widget's state. Programmers can leverage *callbacks* to coordinate effects and events without the need to poll each widget for changes to their state. *Callbacks* are specific to each widget instance. The effect of the *callback* will be dependent on the specific *callback* linked to the widget.

Chapter 5

Implementation

5.1 Overview of the Project

In the following sections, we discuss the implementation of the project with its advantages and disadvantages. Each component of the *engine test cell suite* is outlined, explaining relevance to the project and compliance to the requirements set out in chapter 3.

5.2 Physical Details

As introduced in chapter 1, the *engine test cell suite* produces diagnostic information from the *engine*. This information is read by the *host* connected to the DAQC. Data is transferred over a network connection to each connected *backend*. These *backends* in turn forward the data to their *frontend* to display to an *operator*.

The *frontend* contains widgets, with which the *operator* interacts with to modify the outputs of the *engine test cell suite*. The state of the *backend* is synchronized

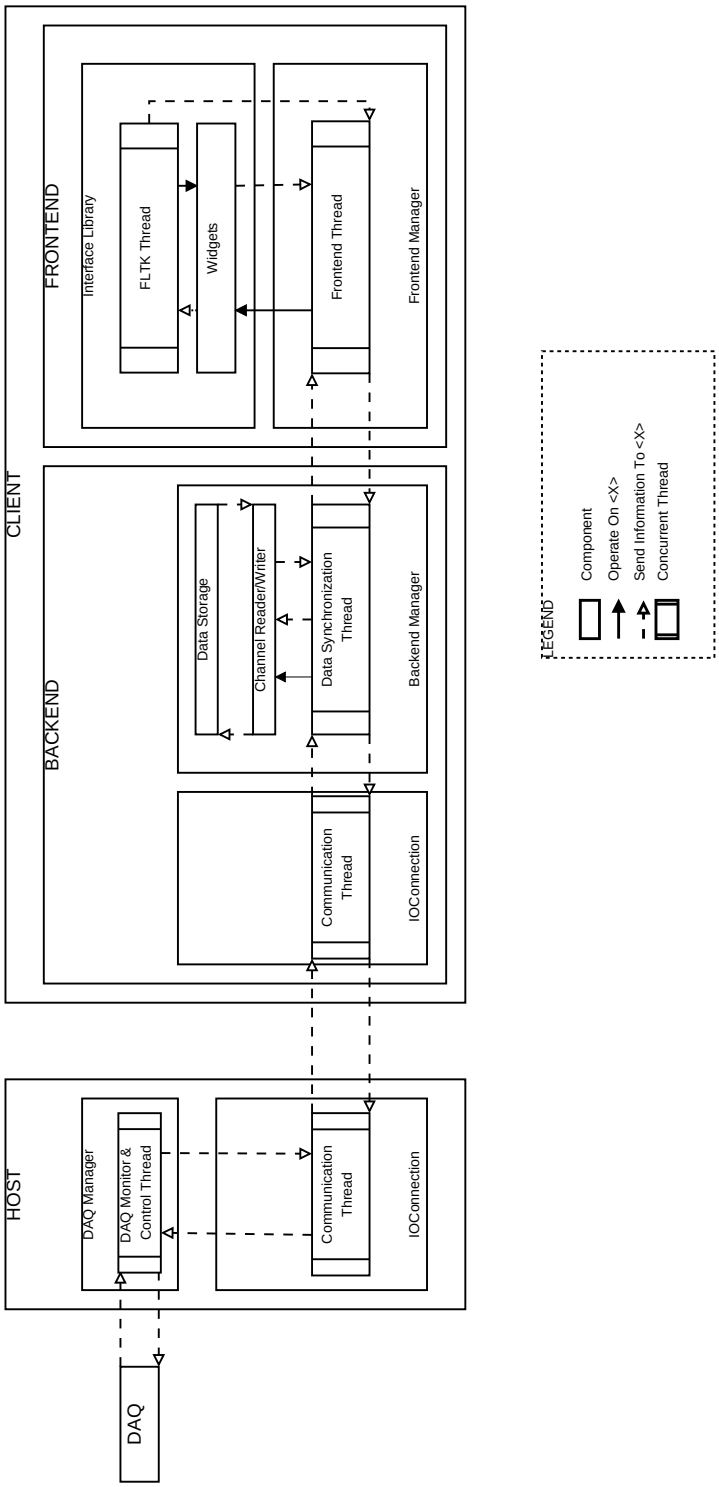

Figure 5.1: Overview of the Engine Test Cell

over the network with other *backends*.

5.3 Engine Test Cell Software

The following sections will expand on the implementation of the *engine test cell suite* components. Focus will be on the communication of data within the *engine test cell suite*, with data being exchanged between the *frontends* and *backends*, and between the *backends* and *host*.

5.3.1 Communication

The *backends* and *host* communicate over the network module using a communication protocol built on top of TCP and UDP. This module is implemented in the IOConnector class.

IOConnector

Information is encapsulated in a communication protocol before being sent within the *engine test cell suite*. The purpose of this encapsulation is to ensure the data being received is complete and valid. The message must not be corrupted during transmission, and the data must be consistent with the specified encapsulation form.

The main procedures required for operating the communication module are:

1. Create connection to specified target data source
2. Push data over communication medium
3. Pop data from communication medium
4. Delete active connection to specified data source

These procedures are defined in polymorphic classes. Each class implements communication over a medium which is used to transfer and receive data. Polymorphism allows for classes to operate in place of a parent class. Each class shares functions and properties from the parent. To allow definition of functions which differ based on the current child class, "virtual functions" are defined to perform a version of the task for the specified class.

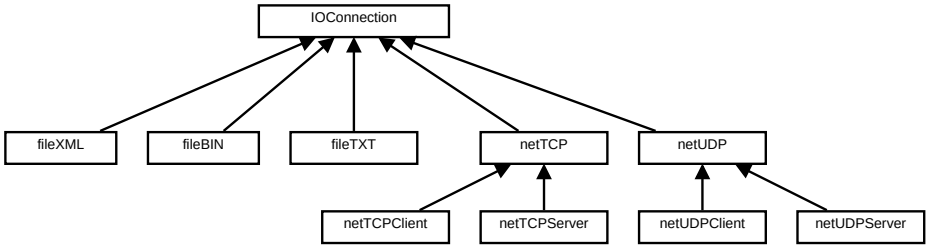

Figure 5.2: Overview of the IOConnection hierarchy

The *engine test cell suite* allows the protocol to use either TCP or UDP as the base network transfer medium. When using TCP, messages are streamed from the sending system to the receiving system. Individual messages are extracted from the stream and parsed in received order. When using UDP, messages are sent as individual packets, which are processed as they are received. UDP does not handle reordering of packets upon receipt. As a result, the protocol must add additional data to timestamp the packets. This ensures data occurring earlier in the state progression of a system does not overwrite a later state.

When a TCP communicator is instantiated, the form will use TCP to communicate data. A UDP communicator will use UDP to communicate. By specifying the operations in derivative classes, changing the medium upon which communications are done is simplified by changing the instantiation type of the runtime class.

Network-based Communicator

The network communicator in the *backend* uses a "many *clients* to one *host*" architecture. While one *host* can send to many *clients* at the same time, *clients* will only communicate with one *host*. The simplicity of this architecture means *client* communicators focus on keeping their connection to the *host* refreshed. In return, the *host* sends to *clients* which are currently active and have not timed out.

Protocols such as TCP have a built-in timeout metric for detecting when a communication target is not available. For communication protocols where a built-in timeout metric is not available (such as UDP), a "timeout-based round robin schedule" is used by the *host* to check for received messages and send messages to *clients*. During a "timeout-based round robin schedule", each *client's* communication channel is observed for incoming messages. When a message is received, the timeout count for that specific *client* is reset. Once the message has been received for that *client*, the next *client* is observed. After all *clients* are observed, the process repeats. During this second phase of the process, the current outgoing message is sent to each *client* as per the schedule. If the maximum specified timeout is reached, the associated *client* is removed from the schedule.

The TCP communicators use "piggybacking" to keep frames together in case messages run over multiple packets. "Piggybacking" is the attachment of one message onto another to avoid sending small messages whereby a large message could accomplish the same purpose. By using a sliding buffer type of "piggybacking", it is possible to maximize the bandwidth between *host* and *client* (Tanenbaum, 2010, 226.p5). If pieces of messages are sent, TCP will reorder them on receival after making sure they are sent successfully. Once reordered, the message is reconstructed and isolated from

garbage data.

In UDP communicators, the messages are sent in one piece up to the maximum transmission unit size (normally 1480–1500 bytes per packet on 100Mbps Ethernet). Due to UDP's lack of retransmission and ordering utility, we cannot use "piggybacking" of data across UDP messages. UDP does not create "communication streams" akin to TCP.

To keep the connection active, the UDP communicator uses a keep-alive packet sent at certain intervals to the UDP *host*. A keep-alive packet is a packet which states to the receiving system that no information is being transferred, only that the transmitting system needs to keep the communication channel active. Each time the *host* receives this packet, it will refresh the duration of communication with the sender. This allows the *host* to consistently send packets to the *clients* without having to emulate the TCP stream handshaking and methods.

To communicate information between files and the *backend*, a "file logging" IO-Connector is used.

File Logging

To log data from an *engine test cell suite*, data is written to file storage. Flat storage files help facilitate this task, but having to create entirely new communicator code would waste the potential for *polymorphism*. Instead of writing an input-output system for files, the communicator has a new derivative class for reading and writing to files of different types.

The file class comes with three different types of communicators:

- Binary Files

- Textual Files
- XML Files

Binary files are file targets in which no transformation of data is done before writing to the file or reading from the file. These files are the fastest to use on systems upon which they were created from. A side effect is an inherent incompatibility with systems using a different binary encoding.

The two byte-orders are "big-endian" and "little-endian". In "big-endian" order, the memory is organized by writing the most signifiant byte first to memory. In "little-endian" order, the memory is organized by writing the least significant byte first to memory.

The incompatibility occurs when data is not translated between byte-orders on two byte-opposite systems. An example would be systems where the byte-order is little-endian versus a system where the byte-order is big-endian. Reading a file written in binary format for one with the other without byte-order conversion may cause the data to be read incorrectly. This is not an issue for textual data, as it is written and operated on in a byte-order agnostic form.

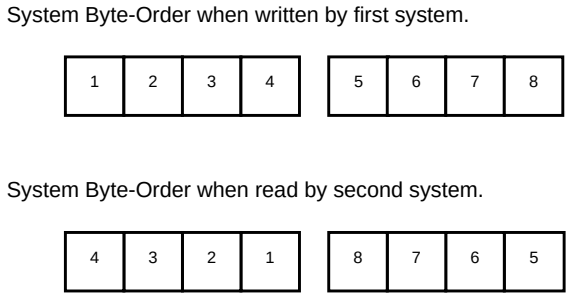

Figure 5.3: Example of writing and reading with two different byte orders.

When textual files are read, they are received "as-is" without parsing for integrated structures and criteria. When writing textual files, the data is written in whichever

form the operating system designates appropriate for text. Data is read one line at a time, up to 1024 bytes. This limitation is due to the restriction of UDP packet sizes in the network interface. Without this restriction, it would require a rewrite of the network UDP and TCP classes to uphold unbound packet and stream sizes.

XML files can be classified under textual files. During implementation however, the library used for XML requires the state of the XML structure to be stored in memory due to hierarchy organization. This leads to a difference between purely textual files and XML files in our system for multiple write-access file targets. For now, only multiple read-access will be touched upon.

XML files are read through the chosen "channel port number". This allows reading different streams of data from the same XML file for different connectors. As long as data is not pushed by both connectors, the system will retain consistency. This is because the state of two separate XML communicators is not shared through the file system until a fresh read of the XML file is executed. If no changes are done to the file, multiple readers from the same XML file will not fail to uphold consistency. The IOConnector fileXML class will read entries from the zeroth entry onwards. As long as another value is available next, the system will continue to queue up entries for reading.

For fileXML-written files, the files will appear as seen in figure 5.4. Reading from channel zero will result in seven values, each of which represents a floating value. When the channel is switched to channel one, there will be three results, each of which results in a textual response. These can be parsed by any component reading from the fileXML IOConnector.

```
<CH0>
        <V0 Value=2.8010 />
        <V1 Value=2.4020 />
        <V2 Value=1.8083 />
        <V3 Value=0.8022 />
        <V4 Value=0.3110 />
        <V5 Value=0.2000 />
        <V6 Value=0.1118 />
</CH0>
<CH1>
        <V0 Value="Reset" />
        <V1 Value="Standby" />
        <V2 Value="Start" />
</CH1>
```

Figure 5.4: Example of XML structure in profiles

Sending and Receiving of Network Data

The sending and receiving of network data is done via TCP and UDP connections underneath the network communicator objects. The communicator classes for TCP and UDP take care of the logistics of network communication facilities, which involve some of the subtleties of the technology. An example of these subtleties can be the "fragmentation", or splitting of packets into smaller sizes, in the TCP protocol.

When receiving a TCP message, the communicator class must wait to see if the entire message has been received completely. This is due to "stream" nature of TCP, which treats the channel as a stream, rather than discrete messages. In some cases, a stream of N bytes can be fragmented into M messages of irregular bytes totalling N bytes in total. Sometimes, two messages can be merged into one message in the stream, which requires detecting the headers of both messages at the end-point for seperation and processing. As a protocol, UDP does not have this situation as messages stay discrete when sent.

Structure of the Network Packet

Transmissions are encapsulated in a network packet. Each network packet consists of an engine test cell suite header, which dictates the start of a packet. The next message component is the size of the body of the message. This size does not include the header or the size portion of the message. The body of the message comes after the header and size portions of the message.

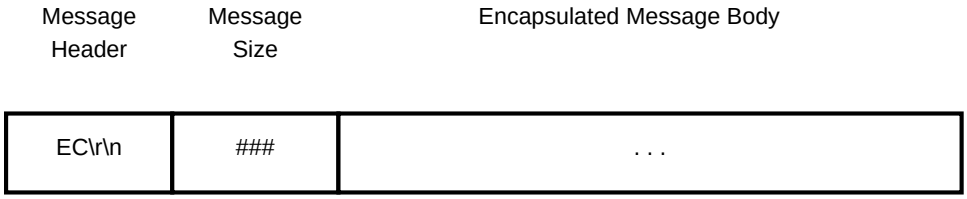

Figure 5.5: Network Packet Structure

Each communicator will organize to receive the message in full before operating on the data. This is important in the case of TCP, where the message can be fragmented into smaller pieces before being received on the receiving end-point as discussed above.

5.4 Frontend/Backend Architecture

During execution, information transmitted to the *client* is received by the respective IOConnection in the *backend*. When received, the data is stored by the *backend* for later reading by the *frontend*. Messages pertaining to channel updates are read, stored in the *backend*, and forwarded to the *frontend's* widget value synchronizer.

The purpose of the widget value synchronizer is to keep all output widgets tracking a channel to follow the value of the specified channel. During execution, this means changes to channels are propagated to and displayed by widgets matching the channel number.

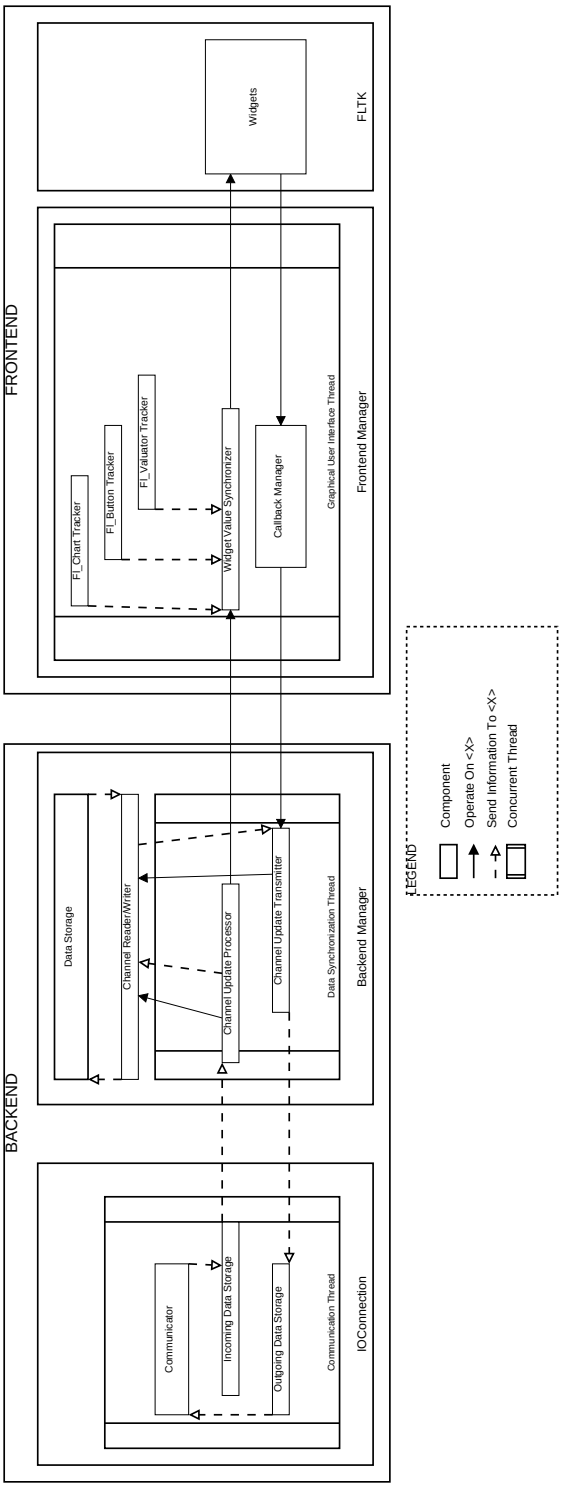

Figure 5.6: Architecture of *frontend* and *backend*

The reverse operation occurs when input widgets tracking a channel have their value altered by the *operator*. When a widget is updated, a callback executes which sends the channel number and update value to the callback manager. The callback manager processes this information by sending the new update to the *backend's* channel update transmitter. The new value is stored by the channel writer and sent to the outgoing data storage. When new outgoing data is available, the respective IO-Connection sends the data to the *host* for modification of the specified channel on the DAQC and further propagation of channel information to other *clients*. This operation is only applicable for the main operating *client*. For all other monitoring *clients*, the update requests sent to the *host* are discarded and an error message is returned to the offending *client*.

With each component of the *client*, threads are used to increase the concurrency of the process.

5.4.1 Thread Concurrency

The process running the *client* consists of four threads:

- IOConnector communication thread
- Backend synchronization thread
- Frontend widget update thread
- FLTK graphical thread

During execution, the communicator thread reads messages from the *host* on the network. Valid *engine test cell suite* messages are placed into an incoming network FIFO queue through a blocking write, to be read by the *backend* synchronization thread. If the incoming network FIFO queue is currently owned by the *backend*

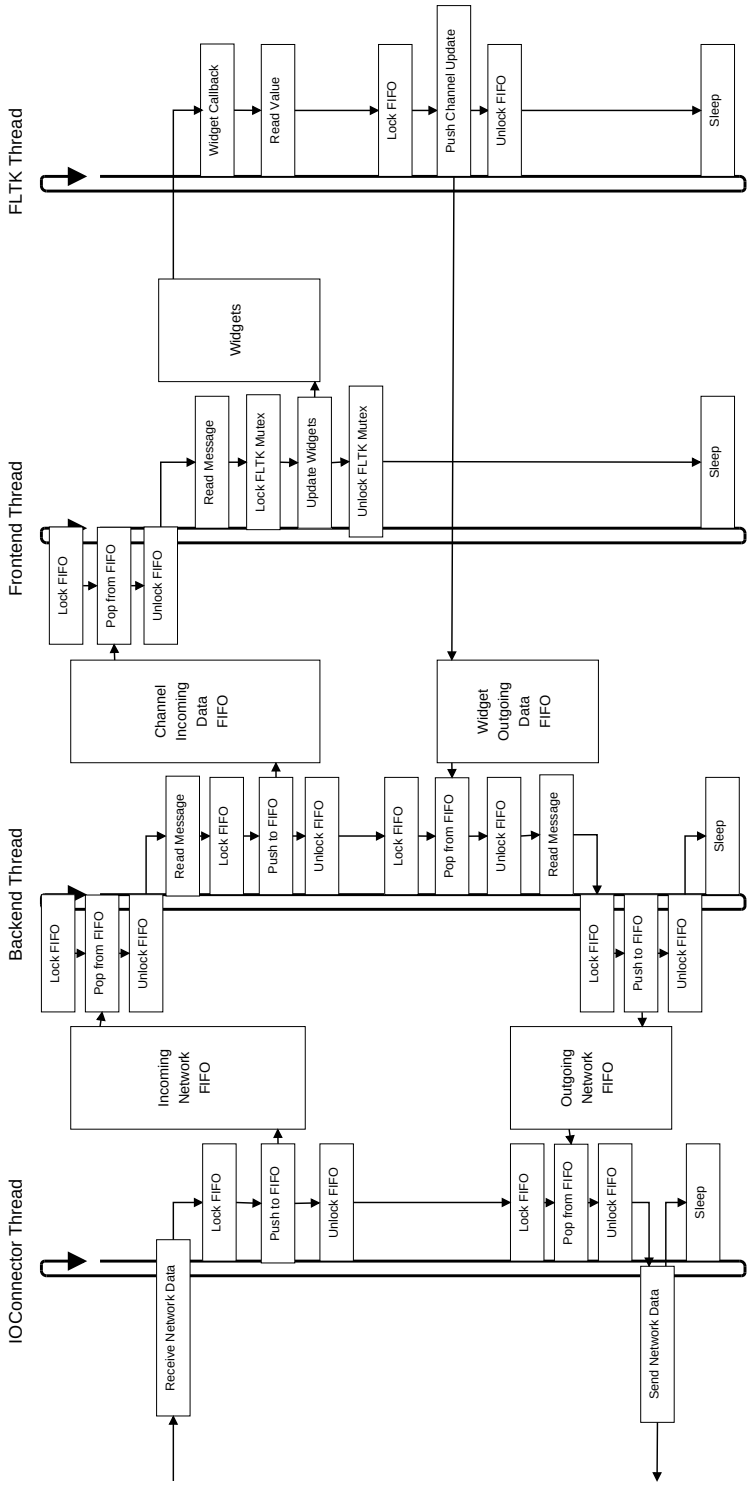

Figure 5.7: Threads of the *client*

thread, the IOConnector thread will block until the *backend* thread has finished checking for new messages on the queue. After checking for a message on the network medium, the communicator thread checks the outgoing network FIFO queue for messages to transmit. If a message is found, the message is sent to the *host*. After completion of a single loop execution, the thread sleeps until the next iteration, after a specified amount of time.

For each FIFO queue used in the *engine test cell suite*, a corresponding mutex is used to enforce blocking and reduce the probability of race conditions between writing to and reading from a FIFO queue. When a lock/unlock applied to a FIFO on figure 5.7, the FIFO being written to or read from by the enclosed push or pop procedure has an associated mutex as specified. These mutexes are local-only, and are not shared nor locked across the medium. This is done to avoid deadlock conditions between the server and clients where one side fails to unlock a mutex during execution. If this were to happen in a shared environment where mutexes are accessible across the medium (whether network, file, or other), it would halt the entire *engine test cell suite*. As the mutexes are local and accessible only through local threads, failure of unlocking a mutex will only affect the running execution of the local system.

The *backend* synchronization thread reads a message from the incoming network FIFO queue. If a message has been retrieved, the message is parsed for new channel value updates. When an update value is detected, the data is placed into the channel incoming data FIFO queue for reading by the *frontend* widget update thread. Once the incoming messages are read, the widget outgoing data FIFO queue is read. For any updates on the queue, these are processed into *host* update messages. The newly generated update messages are placed in the outgoing network FIFO queue for

transmission to the *host*.

The *frontend* widget update thread reads a message from the channel incoming data FIFO queue. If a channel update is available, the channel and value are extracted from the message. From this, FLTK graphical updates are locked, and the widgets matching the channel number are determined. The specified channel and their values are changed to the incoming channel value. Once the widgets have been updated, FLTK is unlocked to allow widgets to be created, altered, and destroyed by the interface library.

The FLTK thread determines if a widget callback has occurred. When a frontend callback occurs, the value of the widget is read and a widget outgoing data message is constructed. This message is pushed onto the widget outgoing data FIFO queue. After the callback has completed, the FLTK thread sleeps until another widget callback or FLTK event occurs.

5.4.2 Interface Library

To generate the user interface, a graphical interface library called "Fast Light Toolkit" (FLTK) is used. The purpose of the interface library is to minimize the code required to create a portable user interface with the ability to operate with data being received from the *engine test cell suite*. FLTK provides the procedures and code required to generate a user interface in an integrated manner with the process.

In FLTK, widgets are designed to respond to the *operator* through callbacks. These callbacks activate when the specified widget has been passed an event instigated by the *operator* through FLTK. In the case of mouse events, clicking on the user interface window will cause a propagation of the event. Each widget found directly

below the mouse cursor is inspected in order of sorting level to see whether it will handle the event. Once a widget accepts the event and completes execution of its handler procedure, the event is "handled", and no further inspection of other widgets is done.

Of the major widget classes in FLTK, the *engine test cell suite* user interface has specific callback procedures for handling the following:

- Fl_Valuator
- Fl_Button
- Fl_Chart

Valuators

Fl_Valuators are widgets which obtain a floating-point value depending on the position of the widget's effector. For widgets without a specified effector, a textual value entry box is available in its stead. A change to this floating-point value occurs through an event, which triggers the widget's callback. This effect can be leveraged to allow synchronization of data as it occurs in lieu of explicit polling of the widget status. When a callback occurs, the *frontend* can pass the information to the *backend* for synchronization.

Buttons

Fl_Buttons are widgets where the callback is activated through mouse interactions by the *operator*. When the callback occurs, an event message is passed by the *frontend* to the *backend* for transmission to the *host*. The value of the button is binary, and does not need to be transmitted for non-toggled buttons. For toggled buttons such

as "radio" and "light" buttons, the value is added to the message explicitly in the callback.

Charts

Fl_Charts are widgets which show a progression of values over time. The *frontend* has a specific tracking procedure for adding new values to widgets reading data from a specified channel. Every additional input event adds a new data point on the chart. The chart keeps track of a specified number of points before the oldest points beyond the threshold are culled from the data pool. This is to reduce the memory burden all Fl_Charts will have on the executing process.

Combined, these three widget classes form the majority of design and implementation of the *frontend*. By creating callbacks and leveraging the synchronization callbacks available in the *frontend*, operation of the *client* is straightforward to design and implement.

Interface Design

In designing and implementing an interface with the *engine test cell suite*, each *operator* input is mapped through valuators and/or buttons in the application. By instantiating a widget from the Fl_Valuator or Fl_Button class and adding it through the *frontend* widget addition procedures, the widgets will be tracked with value and channel by the frontend thread. Outputs are mapped through output valuators and/or charts in the application to display the value of their specified channel.

As stated in section 5.4, when new data is available on the DAQC through changes to inputs, the data is sent to each *client* to update the specified channels pertaining

to the inputs. Once the data has been passed to be read by the frontend thread, any output valuators and/or charts are identified through their channel-to-widget mapping. These specified widgets are updated to match the incoming channel value, and refreshed to show to the *operator* of the change. This freedom to specify channels to widgets from the application directly greatly reduces the amount of redundant settings required on both the *client* and *host* within the *engine test cell suite*.

Channels allow inputs and outputs on the DAQC to be mapped to widgets by *operators* without the need to define explicit widget/channel pairings on the *host*. In the case of DAQC outputs to the *engine test cell suite*, multiple widgets can track the value and display it in specified formats. As an example, the torque being applied by an engine can have the instantaneous value and the history of the value shown simultaneously via an Fl_Valuator and an Fl_Chart. The Fl_Valuator will show the instantaneous value for every received value while the Fl_Chart will show the history of the value over time.

Overall design of the system is simplified by the reduction of data modification and to channels and major widget types. By avoiding pairing of channels to widgets between the *clients* and *host*, the interface can freely track and set channels as necessary. The *host* sets the values as requested if possible, and returns values as specified by the DAQC.

Chapter 6

Validation

Validation is the application of *testing* and *inspection* methods to show whether the system fulfills the requirements it was designed to meet for the stakeholders. Our stakeholders, as stated in section 3.1.2, are the *developers* and *operators* of the *engine test cell suite*. To use these methods, we must first explain what they encompass and how they are beneficial in the *validation* process.

Testing is a method which compares computed output with the intended output of the target. Often this involves the creation of *stubs*. A *stub* is a custom piece of *testing* code made to call procedures in the target component with specified data. The purpose of this is to test whether the component works as defined when confronted with pre-determined situations and requirements. For each input data parameter, the component is expected to operate in a specified manner. For malformed data, the component must compensate and perform in a reliable manner within the guidelines stated by the stakeholders.

Inspection is the comparison of the system at a qualitative level with respect to a checklist of requirements. Through *inspection* we determine whether the system

satisfies the expected requirements. The process is similar to the *walkthrough* specified by the ESA, where segments of documentation or code are examined by reviewers who "ask questions and make comments about possible errors [...] and other problems." (European Space Agency, 1995, A-6.p3).

6.1 Frontend Validation

Validation of the *frontend* is done through *inspection* of the widgets with regards to the requirements specified in sections 4.1.1 and 4.1.2. To validate the *frontend* to *backend* communication, *testing* is done using *stubs* to emulate different states of the *backend* and output *stubs* to show whether the component succeeded or failed the *validation*.

6.1.1 Widget Dependencies

Through *inspection*, we observe that FLTK widgets can inherit properties and procedures from previously defined widgets. These predecessors define properties which are shared by descendant widgets. Descendant widgets are observed to have functionality of a higher specificity than their ancestor widgets. As part of the FLTK library, widgets are self-contained upon runtime creation and initialization.

The condition of widget independence is valid through the FLTK library.

6.1.2 Widget Value Display

For *testing* the acceptance and restriction of input values, a series of widgets have the following requirements applied:

1. Widget pair will be instantiated on the *frontend*.
2. *Frontend* will transmit all values written by the primary widget to a *backend stub*.

The values received are read to determine whether the values sent by the *frontend* match the originating values in the widget.

In the second phase of widget value *testing*, a *backend stub* is applied to the tested components to execute the following:

1. *Backend stub* transmits "values read" to the *frontend*.
2. Widget receive value from the *frontend* and display value.

Values entered into widgets must not convert the value before being resolved by the *backend*. Input fields can restrict values to certain types, but it must not implicitly convert the values within the widget before sending to the *backend*.

Restriction of values is part of the FLTK library. Widgets will accept or restrict values depending on the input method used for reading and writing values.

Testing was done by creating a *stub* to call the initialization procedures of the widgets and the setup of the *backend* channels.

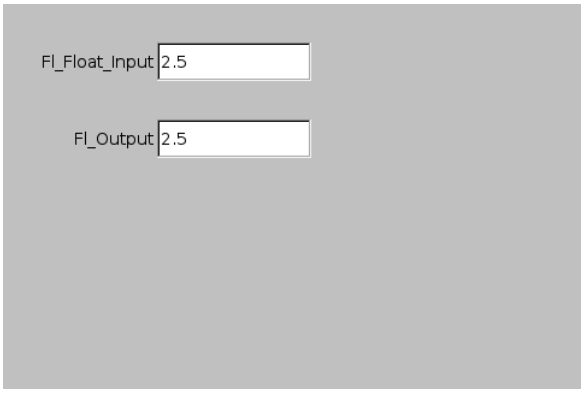

Figure 6.1: Widget value before interaction.

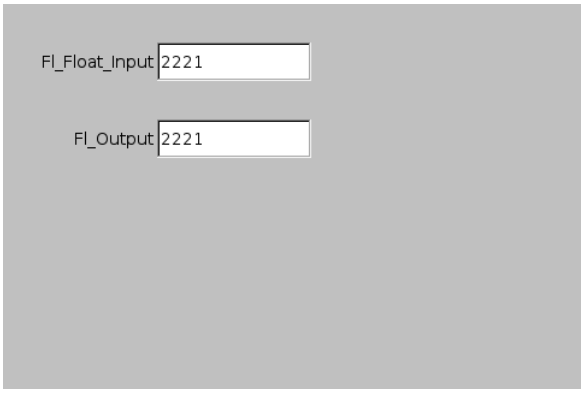

Figure 6.2: Widget value after interaction.

As the output value is the same as the specified input value, *testing* validates the system has operated correctly for values sent through the *backend*.

6.1.3 Widget Availability

The use of widgets requires each widget to be "available" for interactions with regards to FLTK event calls. Availability is defined to be the property of a widget in which it can be interacted with by an *operator* through events. As stated in section 5.4.2, events trigger callbacks of available widgets to perform their specified task. When a widget is unavailable, callbacks are not triggered for the specified widget.

When reading and accessing a widget's data (i.e. value, position, etc) from multiple threads, the reading and writing are considered critical sections of execution. As stated in section 2.6.1, for critical sections a lock is applied to the shared resource to avoid race conditions. To keep the interface available, locks on critical sections inside widget callbacks must finish execution in a specified length of time without deadlocking.

Through inspection of the widgets on the *frontend* with data being read from the

backend, we note that the interface is locked before changing the values of the widgets associated to the incoming channel data. The interface is unlocked once the values have been applied to the widgets.

With regards to the time required to perform the value assignment during the lock, the average time for the *engine test cell suite* executing was 1.9040×10^{-6} seconds, with a worst execution case of 4.6596×10^{-4} seconds as an outlier at the 22901st index out of 42324 locks. The standard deviation of the lock timings was 2.9463×10^{-6} seconds. During testing of the mutex locks, messages were received by the *client* from the *host* at a rate of 1000 messages per second over a test of approximately 30 seconds.

In contrast to the frequency at which changes are done to the *frontend* by the *operator*, on the order of 0Hz to 5Hz, the lock times are negligible.

This *inspection* shows that the widgets stay available as per the aforementioned requirements.

6.1.4 Frontend-to-Backend Communication

As specified in section 5.4.1, information travelling from the *frontend* to the *backend* uses a FIFO queue for updating all widgets matching the specified channel of the outgoing information. The *backend* uses this queue to prepare for sending channel updates to the *host* through the IOConnector thread.

In testing the communication between the *frontend* and *backend*, a pair of widgets (with communication *stubs*) are used to show the information received after being sent from one *frontend* widget to another on the same specified channel. A "value input" *stub* widget takes the *operator's* floating point input and transmits it to the *backend*.

Once the *backend* has received the value, it updates the channel and informs the *frontend*. A "value output" *stub* widget was assigned to the same channel, which updates when the *backend* informs the *frontend* of a successful change. The value of the *stub* was compared to the input value to determine if the component was tested successfully.

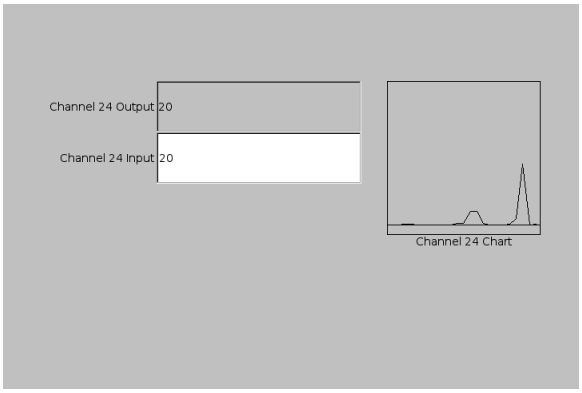

Figure 6.3: Widget values after retrieval from *host*.

In figure 6.4, the *host* outputs the channels verbosely as they are updated. Each channel updates as the values are changed to illustrate successive values between each *frontend*. From one *frontend* the values are created and sent to all others.

```
main.cpp:: Channel 24 changed to 1.000000.
main.cpp:: Channel 24 changed to 2.000000.
main.cpp:: Channel 24 changed to 20.000000.
main.cpp:: Channel 24 changed to 2.000000.
main.cpp:: Channel 24 changed to 23.000000.
main.cpp:: Channel 24 changed to 2.000000.
main.cpp:: Channel 24 changed to 0.000000.
main.cpp:: Channel 24 changed to 5.000000.
main.cpp:: Channel 24 changed to 50.000000.
main.cpp:: Channel 24 changed to 500.000000.
main.cpp:: Channel 24 changed to 50.000000.
main.cpp:: Channel 24 changed to 5.000000.
main.cpp:: Channel 24 changed to 0.000000.
main.cpp:: Channel 24 changed to 2.000000.
main.cpp:: Channel 24 changed to 22.000000.
main.cpp:: Channel 24 changed to 223.000000.
main.cpp:: Channel 24 changed to 2233.000000.
main.cpp:: Channel 24 changed to 2.000000.
main.cpp:: Channel 24 changed to 20.000000.
```

Figure 6.4: Channel values on the *host*.

As data is received in a manner which matches the transmitted data, the test is

shown to be correct.

6.2 Backend Validation

The *backend* is validated through inspection of the components which provide functionality of physical protection states and data transceival between the *frontend*, itself, and the *host*.

6.2.1 Functionality

The functionality of the *backend* must comply with the specified physical protection and virtual observation functions. The following outlined features are required as per sections 4.2.1 and 4.2.2.

Emergency Stop

To test the emergency stop of the *backend*, an explicit ESTOP procedure is called through a *frontend stub*. The *backend* has an explicit function SendESTOP(). When called, SendESTOP() will create a message to the *host* to reset channels to their defaults and disable all future writes. If the main operating *client* has sent the request, or forwarded it from any of the monitoring *clients*, the *host* will enter a state where any attempts by the *frontend* to alter values will be discarded by the *host*. Changes in this state will not be propagated to any other *backends* until the *backends* receive a restart message.

A *host stub* is used to show the point at which an "emergency stop" was put into effect. This would occur when the main operating *client* transmitted the ESTOP

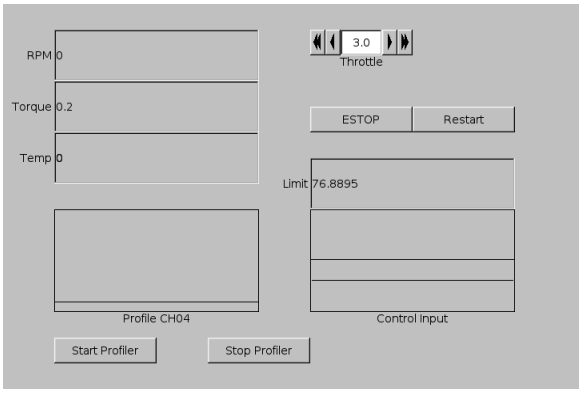

Figure 6.5: Final state of the *frontend*

```
main.cpp:: Channel 24 changed to 223.000000.
main.cpp:: Channel 24 changed to 2233.000000.
main.cpp:: Channel 24 changed to 2.000000.
main.cpp:: Channel 24 changed to 20.000000.
main.cpp:: EMERGENCY STOP.
main.cpp:: EMERGENCY STOP
main.cpp:: Cannot write. System Halted.
main.cpp:: Cannot write. System Halted.
main.cpp:: Restart Detected.
main.cpp:: Restart Detected.
main.cpp:: Resume has been established.
main.cpp:: Restart Called.
main.cpp:: Channel 7 changed to -0.100000.
main.cpp:: Channel 7 changed to 0.000000.
main.cpp:: Channel 7 changed to 1.000000.
main.cpp:: Channel 7 changed to 2.000000.
main.cpp:: EMERGENCY STOP.
main.cpp:: EMERGENCY STOP
main.cpp:: Cannot write. System Halted.
```

Figure 6.6: Status messages shown on the *host*

message and the *host* received the message. In Figure 6.6, the *host stub* declares the system to have reached an emergency stop. The attempt to switch the throttle in channel 7 to 3.0 is met with a message about the system being halted. To restart the *host*, a SendRestart() is used by the main operating *client*. This function is available during an emergency stop. When the *backend* has been called using this function, it will send a message to the *host* requesting a restart. If the *engine test cell* is in the stopped state, the *host* will restart in the default state and allow the *engine test cell* to restart testing.

The *host stub* received the "restart", followed by changes to the 7th channel. After the changes were observed, an "emergency stop" was sent again to see whether the system could be safely restarted. No values could be changed while the system was under the "emergency stop" state.

One observation is that the system only requires the singular main operating *client* to operate the restart. This will be discussed in the future considerations section 7.2.

Through inspection, this test shows the emergency stop working as intended.

Physical Limits

Physical limits of the channels is a design consideration as stated in section 4.3.1. To show the system's physical limits controls are adequate and operational, *testing* is done to the *host* through a stub *client*.

To test the physical limits management of the *host*, the output limiter flags must be activated for the channels to be tested. These flags are tested on the *host* to determine the limit function, as defined in the *host* program, to be applied to the specified channel. The test client sends updates to the *host* while receiving channel

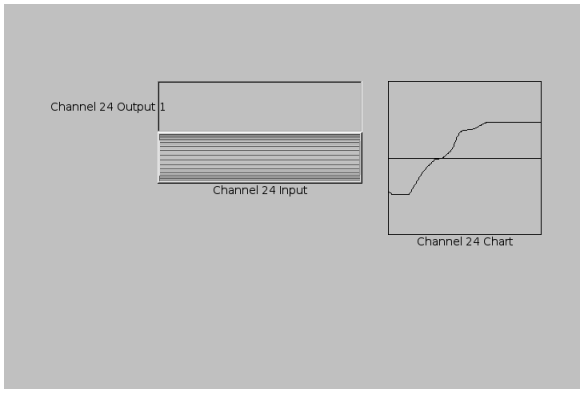

Figure 6.7: Limit (-1.0,1.0) applied to channel 24.

updates in return. The *host* must limit the channels to a specific range of values as determined by the connected device physical limits.

An example of the limits of a channel was channel 24, where the values were restricted on the *host* to stay within the range of [-1.0, 1.0].

A *stub* is used to transmit values for channel 24 to the *host*. Channel 24 is inserted as a testing *stub* to return the current value during execution. As the roller *stub* widget was operated to try and exceed 1.0 towards the positive direction, the channel *stub* produces only 1.0. As the roller *stub* widget was operated to try and exceed -1.0 towards the negative direction, the channel *stub* produces only -1.0.

As shown in Figure 6.7, this test validates the physical limiters to be working as intended.

6.2.2 Information Transmission

The reliable transfer of information from the *backend* to the *host* is required for the transmission of control data. The following sections show the results of the requirements as stated in section 4.2.3.

6.2.3 Network Functionality

As a major constituent of the *backend*, the network communicator forms a synchronization between all *engine test cell suite* clients. Forming a reliable backbone of the *engine test cell suite* starts with being reliable in the existence of errors. The two most common forms of malformed network information is incoming errors, and loss of connectivity to the *host*.

Gracefully Discard of Incoming Errors

The *host* and *client* must work reliably in the presence of malformed network data. In *testing* this condition, a network *stub* transmits malformed data to the *host* and *client* while a network *stub* transmits to the receiving system with correctly formed data.

To test the *client* and *host* for working reliably in the presence of malformed network data, two network *stubs* are used to send data to the specified receiving system. One *stub* sends a series of malformed data messages while the other *stub* sends a series of correctly formed data messages. During the test, a series of 1000 messages are transmitted to the *client*. Of these 1000 messages, 50 messages contain correctly formed data. The other 950 messages contain malformed data. After the test, a *stub* retrieved the number of correctly formed messages retrieved by the *client*. *Testing* is done with each of the IOConnection network communicators to determine reliability.

For each IOConnection network class, the number of correctly formed messages retrieved from the 1000 messages was 50 messages. Through testing, the network classes performs with reliable communication between the *client* and *host* working as

expected.

Automatic Reconnection on Connection Loss

To test the automatic reconnection on connection loss, the *host* and *backend* are detached from each other for an extended period of time before being allowed to join the network again. After a connection to the *host*, the *backend* must keep the connection active under the specified supported protocol. As the underlying mechanisms are different for each protocol, we focus on the implemented recovery procedures for each.

Under TCP, the *backend* will test if the *host* has returned to online status. For such an event, the *backend* proceeds to create a new valid connection to the *host* and resume sending of backlogged data.

Under UDP, the *backend* will send "keep-alive" packets to the *host*. When receiving a valid "keep-alive" packet with proper data structure, the *host* adds the valid *client* to its sending pool and sends new updates to it.

During testing of the TCP connector, the *host* was forcefully removed from the network. The *client* switches to recovery mode, which causes all new packets to be stored in the queue while waiting for the connection to be re-established. After the *host* is brought back online, the *client* reconnects successfully and sends all stored packets to the *host*.

During testing of the UDP connector, the *host* was forcefully removed from the network. As the protocol itself is unreliable, the system continues to send to the *host*. After the *host* is returned to online status, it begins receiving the "keep-alive" packets, and re-adds the *client* to the transmission list. After this state, the *client*

starts receiving from the *host's* data stream once again.

This test shows the network automatic reconnection functionality to be working as intended.

6.3 Validation Conclusion

Each *engine test cell* component showed functionality as dictated by the requirements set out in the *design* stated through section 4. For components residing within the FLTK included library, the functionality was inspected to show compliance with the requirements. Through both *inspection* and *testing*, the validity of the components within the *engine test cell* could be shown and conveyed.

Chapter 7

Conclusion and Future Work

7.1 System Conclusion

In this thesis, we developed an engine test cell suite to gather quantifiable data from an engine. To enable remote operation of the engine test cell, we designed the system with two process types; the host and the clients. The host process reads output data from the DAQC to send to the clients while reading input data from the main operating client to send to the DAQC. The clients receive output data from the host to display to the operators while reading input commands from the operator on the main operating client. Operator data is processed and sent as input data updates to the host.

The engine test cell suite is composed of a single host process and at least one client process. Each client process connects to the host to obtain data and listen for updates from the main operating client. Data is obtained from the DAQC connected to the engine, which is read by the host and sent to the clients. Each client contains a backend thread, frontend thread, FLTK thread, and network communicator thread

to take advantage of multitasking capabilities of the operating system executing the client.

During execution of the engine test cell suite, the client and host processes execute as a set of threads. The IOConnector thread handles network communication to send communication data to and receive data from the host. The backend thread handles incoming channel synchronization messages from the IOConnector's incoming network FIFO queue, in addition to creating and pushing new channel synchronization messages to the IOConnector's outgoing network FIFO queue. Any messages pertaining to synchronization from the IOConnector on the backend are processed, and changes to the channels are pushed to the frontend thread's channel incoming data FIFO queue. The frontend thread maps data channels to user interface widgets. Any changes to the channels pushed to the channel incoming data FIFO queue is read by the frontend thread and the specified channels are set to the new value. The FLTK thread displays the widgets on the user interface and handles any inputs given by the operator through mouse and keyboard inputs. When an event causes a mapped widget to alter its channel value, the FLTK thread pushes the new value and specified channel to the widget outgoing data FIFO of the backend.

Communication is handled by a thread running within the host process and within each client process. Each client communication thread connects to the host to read incoming output data to push on to the backend's incoming network data FIFO queue. Any messages to be sent by the backend are pushed onto the communication thread's outgoing network FIFO queue.

The use of networked communication allows monitoring of the engine from any networked location. Showing active systems executing in presentation scenarios and

other demonstrative purposes are definite possibilities. With reliable data transfer for control messages, the distance for which monitoring can be granted is limited to the reliability of the network infrastructure and the latency between the host and clients.

7.2 Future Considerations

For future work, this project will require work to be done on the IOConnector's fileXML communication class to facilitate multiple writes to the same target file. Reading using multiple communicators on the same file is does not cause corruption of the file, but multiple writers will overwrite it with malformed data. This is due to the TinyXML library loading the file into memory during execution. In addition to the fileXML class, the netUDP and netTCP communication classes require support to send messages through a sender-exclusion path. The data sent by one network communicator should not be receiving a copy of the same data to itself.

In terms of the IOConnector's system error condition monitoring, the system must be changed to return error codes for all operations which return a new object. If error codes are not acceptable for future implementation choices, C++ exceptions could also be used to throw when errors are determined to have occurred. For this to be consistent, the entire IOConnection framework must be updated to match the new error condition monitoring.

Security of a system is a major concern in protecting the *operators* and other workers from malicious entities trying to create a hazardous environment. One method to minimize the risk from such entities would be to create a pair of encapsulated IOConnectors using Transport Layer Security with a replacement schedule for removing compromised private keys from the systems. With these protections, trying to send

compromised messages to the *host* would require breaking the security or finding a compromised private key. Even with the protections available, a malicious internal *operator* could attack the system from an authorized *client*, which would bypass the security without needing to break it. As it stands, the security of the system would require a hazard analysis into the types of risks and concerns during the engine test cell's operation.

Finally, implementation of a widget factory and widget state replicator would allow on-the-fly online modification of the *frontend* during execution. This would make changes done to the *frontend* require little to no downtime. Additions are necessary to the *backend* message parser to react to such messages and push widget positional and instantiation data to the *frontend*. The *frontend* would need to read these changes into the widget state replicators to copy and adjust the widget state of the *frontend* between each *client*.

Overall, the project has led to the creation of an *engine test cell suite* without requiring LabVIEW. With further work, this system could be improved upon to create a fully featured system to be used in many testing facilities covering a wide variety of instrumentation tasks.

Bibliography

Association, M. (2012). Modelica ® - a unified object-oriented language for systems modeling language specification.

Branicky, M., Phillips, S., and Zhang, W. (2000). Stability of networked control systems: Explicit analysis of delay. In *Proceedings of the American Control Conference*, pages 2352–2357.

Centre for Mechatronics and Hybrid Technologies (2013). Engine test cell. Cam Fisher.

Cook, S. (1989). Introducing object-oriented systems. In *Applications of Object-Oriented Programming, IEE Colloquium on*, page 1/1.

Edwards, M. L. (2001). S-paramaters, signal flow graphs, and other matrix representations.

European Space Agency (1995). Guide to software verification and validation. `ftp://ftp.estec.esa.nl/pub/wm/anonymous/wme/bssc/PSS0510.pdf`. [Online; accessed 25-October-2013].

Galitz, W. O. (2002). *The Essential Guide to User Interface Design: An Introduction*

to GUI Design Principles and Techniques. John Wiley & Sons, New York, NY, 3rd edition.

Galloway, B. and Hancke, G. P. (2012). Introduction to industrial control networks. *Communications Surveys & Tutorials, IEEE*, **PP**(99), 1–21.

Halpern, J. Y. and Moses, Y. (1990). Knowledge and common knowledge in a distributed environment. *J. ACM*, **37**(3), 549–587.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274.

Harel, D. (1988). On visual formalisms. *Commun. ACM*, **31**(5), 514–530.

Huang, F. (2010). *State Diagrams: A New Visual Language For Programmable Logic Controllers*. Master's thesis, McMaster University, Hamilton, Ontario, Canada.

IEEE (2008). IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–58.

IEEE (2013). IEEE-SA -IEEE get 802 program. `http://standards.ieee.org/about/get/802/802.3.html`.

James, D. (1990). Multiplexed buses: the endian wars continue. *Micro, IEEE*, **10**(3), 9–21.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, **21**(7), 558–565.

Mathworks (2013). Solve fully implicit differential equations, variable order method - matlab ode15i. `http://www.mathworks.com/help/matlab/ref/ode15i.html`.

Parnas, D. L. (2003). Requirements documentation: A systematic approach. Technical report, University of Limerick.

Rabin, M. O. and Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development*, pages 114–125.

Rajagopalan, A. and Washington, G. (2002). Simulink tutorial. `http://mercur.utcluj.ro/mobile/cursuri_oltsi/SimulinkTutorial.pdf`. [Online; accessed 05-February-2013].

Schwarz, R. and Mattern, F. (1994). Detecting causal relationships in distributed computations: in search of the holy grail. *Distrib. Comput.*, **7**(3), 149–174.

Smith, C. (2012). A powerful new tool for UI programming–user interface event programming. [Online; accessed 25-January-2013].

Tanenbaum, A. S. (2010). *Computer Networks*. Prentice Hall, 5th edition.

Wills, C. (1994). User interface design for the engineer. In *Electro/94 International. Conference Proceedings. Combined Volumes.*, pages 415–419.

Xudong, L. and Jiancheng, W. (2007). User interface design model. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 3, pages 538–543.