A Distributed Component-based Software Framework for Laboratory Automation Systems

by

Venkataramanan Kuppuswamy

A Thesis Presented in Partial Fulfillment of the Requirements for the Degree Master of Science

Approved November 2012 by the Graduate Supervisory Committee:

Deirdre Meldrum, Co-Chair
James Collofello, Co-Chair
Hessam Sarjoughian
Roger Johnson

ARIZONA STATE UNIVERSITY

December 2012
ABSTRACT

Laboratory automation systems have seen a lot of technological advances in recent times. As a result, the software that is written for them are becoming increasingly sophisticated. Existing software architectures and standards are targeted to a wider domain of software development and need to be customized in order to use them for developing software for laboratory automation systems. This thesis develops an architecture that is based on existing software architectural paradigms and is specifically tailored to developing software for a laboratory automation system. The architecture is based on fairly autonomous software components that can be distributed across multiple computers. The components in the architecture make use of asynchronous communication methodologies that are facilitated by passing messages between one another. The architecture can be used to develop software that is distributed, responsive and thread-safe. The thesis also develops a framework that has been developed to implement the ideas proposed by the architecture. The framework is used to develop software that is scalable, distributed, responsive and thread-safe. The framework currently has components to control very commonly used laboratory automation devices such as mechanical stages, cameras, and also to do common laboratory automation functionalities such as imaging.
DEDICATION

To my parents, family, friends, and colleagues.
ACKNOWLEDGEMENTS

I am very grateful to Professor Deirdre Meldrum, Dr. Roger Johnson and Mr. Dean Smith of the Center for Biosignatures Discovery Automation (CBDA) at the ASU Biodesign Institute for giving me an opportunity to work with their team and also for their guidance in this thesis and research. I would also like to thank Professor James Collofello and Professor Hessam Sarjoughian for their guidance in this thesis and research. I also thank the engineers at the Microsoft Robotics Studio team for their help and support in answering questions related to Microsoft Robotics Studio.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vi</th>
</tr>
</thead>
</table>

## CHAPTER

1  **INTRODUCTION**

   - PREVIOUS WORK ................................................................. 6
   - CONTRIBUTIONS OF THE THESIS ............................................ 14

2  **CODA**

   - BASIC CONCEPTS AND DEFINITIONS ........................................ 16
   - COMPONENT HIGH LEVEL DESIGN ............................................ 20
   - REQUIRED BEHAVIOR, OPERATIONS AND PROPERTIES .................... 24
     
     - Required Behavior .......................................................... 25
     - Required Operations ........................................................ 28
     - Required Properties ....................................................... 28
   
   - COMPONENT USE AND LIFETIME MANAGEMENT ............................. 29
     
     - Basic Concepts and Definitions for Component Use .......... 29
     - Interface Object ............................................................ 32
     - Component Lifetime Management ....................................... 34

3  **CODA FRAMEWORK** .............................................................. 36

   - CONCEPTS USED ................................................................. 36
     
     - UML State Machines ....................................................... 36
     - Microsoft Robotics Studio .............................................. 39

   - COMPONENT DESIGN ........................................................... 43
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Component High Level Design</td>
<td>20</td>
</tr>
<tr>
<td>2. Action Execution</td>
<td>23</td>
</tr>
<tr>
<td>3. Task Execution</td>
<td>24</td>
</tr>
<tr>
<td>4. Component Behavior FSM</td>
<td>27</td>
</tr>
<tr>
<td>5. How the Interface Object Works</td>
<td>34</td>
</tr>
<tr>
<td>6. Component Design</td>
<td>43</td>
</tr>
<tr>
<td>7. CodaComponent Design</td>
<td>45</td>
</tr>
<tr>
<td>9. DSS Component Lifecycle</td>
<td>48</td>
</tr>
<tr>
<td>10. Client-Local Component Communication</td>
<td>49</td>
</tr>
<tr>
<td>11. Client DSS Component Communication</td>
<td>50</td>
</tr>
<tr>
<td>12. Stage Controller Abstraction FSM</td>
<td>51</td>
</tr>
<tr>
<td>13. Camera Abstraction FSM</td>
<td>52</td>
</tr>
<tr>
<td>14. Test Component</td>
<td>54</td>
</tr>
<tr>
<td>15. IMAQdx Camera Component Test</td>
<td>55</td>
</tr>
<tr>
<td>16. Stage Controller Component Test</td>
<td>56</td>
</tr>
<tr>
<td>17. Imaging Component Test</td>
<td>59</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Laboratory automation involves designing and developing automated systems that perform processes that are otherwise performed manually, or enable experimentation that otherwise would be impossible. Automation can significantly increase the throughput of a laboratory, offloading labor-intensive tasks, thereby increasing the productivity of the researchers. It has become imperative for today’s laboratories to apply technology to achieve timely progress and also to remain competitive. Even the roadmap of NIH identifies technology development as one of its mission critical factors [1]. A significant number of laboratories are moving towards that goal by automating more and more tasks using a number of advanced technologies [2]; hence, software developed for these kinds of systems is also becoming very sophisticated, and as a result, special emphasis needs to be given when developing this software.

Typically, software for a laboratory automation system involves controlling and coordinating the use of multiple, different types of devices, in order to automate an operation or procedure. Examples of devices that usually require automated control include motion control stages, cameras, lasers, pumps, and valves. Devices have their own memory and processing units, and work independently from the computer that they are connected to. Normally the manufacturer of a device provides its own device driver; as a consequence, programming software for a specific device from a manufacturer makes it very difficult to extend and maintain. Software for a laboratory automation system should be designed and developed in such a way that it is easily extensible, allowing the incorporation of new features and
devices from various manufacturers. Also, the software should not require modification when a particular type of device is replaced with the same type of device but from a different manufacturer (use of software abstractions).

Responsiveness of software is its ability to give timely feedback to its users as to what its current status is, as well as what operations it may be performing [3]. The feedback can be in the form of a progress bar, acknowledgment message, or completion message. Responsiveness can greatly increase the usability and robustness of software. Software that lacks responsiveness usually does not respond to user inputs in a timely manner, nor does it update the user on the software’s status in a timely manner - something which is unacceptable in a robust automation or control system. Special emphasis needs to be given to responsiveness when developing software for laboratory automation systems since such software must have a near real-time capability of controlling and communicating with laboratory devices. Operations involving laboratory devices such as motion control stages, cameras and lasers can be extremely slow compared to the time it takes to process data. This is because the automation software may incorporate mechanical devices that must perform physical actions, such as opening and closing of the shutter in a camera, or repositioning a stage; which is usually orders of magnitude slower than the time it takes to process information and effect program flow or control. If these operations are performed synchronously, then the application is giving up the control of the thread’s processing while a device like a camera, or stage controller, performs its operation; and this makes the responsiveness of the application highly undesirable, and even worse: unpredictable. To handle this problem,
operations on devices should be done asynchronously, freeing up the application thread to handle user requests and status updates.

Since software for laboratory automation systems require responsiveness, and often time, its components are called upon to perform multiple operations simultaneously; the components should be programmed to exhibit asynchronous behavior using multithreaded programming concepts. In a multiprocessor computer, the threads can be distributed evenly across multiple cores to improve scalability and performance. Since multithreading often involves access and update of shared data, it introduces problems of thread-safety and deadlocks. Thread-safe code is code that maintains the consistency of shared data even when called from more than one thread simultaneously. Programs that have code that is not thread safe can cause errors that are notoriously hard to diagnose and debug, as well as exhibit unpredictable behavior. Thread-safety is usually achieved through the use of synchronization mechanisms such as locks [4]. Unfortunately, locks hinder performance, and when used carelessly they also cause deadlocks. A deadlock is a situation where two or more competing threads are each waiting for the other to finish, and thus neither one ever does [5].

The software for a laboratory automation system is usually divided into various components, and each component may run optimally only under a specific operating system and hardware. For example an imaging component may work optimally in a high speed 64-bit operating system because it performs a lot of computation and data analysis, whereas a device control component may work optimally in a 32-bit Operating System, since device drivers are usually more stable and reliable in 32-bit operating systems. As a result, having the entire software for the system operating on a single
computer, which has a single operating system and a single set of I/O hardware, can make it difficult for the code to work effectively, and in many cases is highly undesirable. In such scenarios, a better approach is to have the software components of the automation system distributed across multiple computers over a network. The automation systems’ software should be designed in such a way that there is flexibility in trying out various component distribution scenarios, in order to choose the best one. The components of a distributed system communicate with each other over the network using message passing. Message passing is a mechanism for inter-process communication that involves sending and receiving messages between processes either on the same computer or on different computers [6]. Another mechanism for inter-process communication is shared memory, which involves multiple processes sharing a piece of Random Access Memory (RAM) to communicate with one another [7]. Shared memory cannot be used in a distributed systems scenario, and synchronization is usually achieved by using locks, which as discussed earlier, has several draw-backs.

It is becoming increasingly important for the software for a laboratory automation system to be extensible, distributed, scalable and responsive. At the same time, the software should also be thread-safe and deadlock free. The software for laboratory automation should also take into account the nuances involved in communicating with devices. These problems become even more significant when designing and developing software for sophisticated automation systems, whose software usually has multi-level compositions and interactions of components. Furthermore, software for such systems has to take into account the issue of convoluted program flow control between multiple components that are present in any reasonably complex
system. The challenges involved in developing software for sophisticated laboratory automation systems makes it imperative that its development be driven by a sound architecture that addresses all of these challenges, and proposes effective solutions.

Added to designing and developing an architecture, the development of a coherent and well thought out framework that is based on an architecture that addresses these issues and requirements will add significantly to the quality and robustness of automation software, by providing infrastructure that ensures that the software developed using it adheres to the aforementioned requirements. This infrastructure will take care of providing the low level code needed to implement the mechanisms that provide capabilities such as: asynchronous behavior, thread safe access to module data, distributed components, and abstractions for common laboratory automation functionality, just to name a few. This in turn, frees up the developer from the time consuming task of dealing with the nuances of developing low level code needed to implement all of these requirements – which in the end, is just boiler plate code that all well designed laboratory automation software should have – to allow more time for designing the aspects of the software that are actually unique to the task at hand. In addition to providing the overall benefit of this infrastructure, the use of a framework will enforce the framework’s design pattern, in effect producing what is known as inversion of control. Far from being just a code library, where the developer can randomly pick and choose what functions or objects to use; the framework will enforce the design pattern that the thesis has come up with, which abstracts out much of the basic common functionality of all laboratory automation software. This allows in effect, the framework to
provide the default behavior for the system, which the developer can then build upon.

One main criticism of frameworks is the amount of time it takes for developers to learn to use them in their projects. However, once the developers have progressed beyond the initial learning curve, future projects can be developed much quicker and with much less effort - and with the numerous benefits that the framework brings with it. From the developer’s perspective, software frameworks consist of what are known as frozen spots and hot spots [2]. Frozen spots of a framework are those that define the overall architecture of the software system, and consist of its basic components, as well as the relationships between them; since all of these parts of the framework remain unchanged, they are known as the frozen spots. On the other hand, hot spots represent those parts of the framework where the developers add their own code, in order to add functionality specific to their own project. Hot spots are usually defined by software frameworks in the places of the architecture where application programmers are meant to make adaptations, if needed.

**PREVIOUS WORK**

The most common architectural paradigms for developing reusable distributed software applications are Distributed Object Architecture (DOA), Component-based Architecture (CBA) and Service Oriented architecture (SOA). DOA involves dividing the software into tightly coupled objects that interact with each other. The objects implement interfaces and can be distributed across different computers thereby providing scalability. Component Oriented Architecture came after DOA and involves dividing the software into autonomous components that interact with one another. Unlike
the objects in DOA which are tightly coupled with each other, components in CBA are loosely coupled with one another. Like the objects in DOA, the components in CBA implement interfaces and can be distributed across multiple computers thereby providing scalability. Service Oriented Architecture came after CBA and involves dividing the software into loosely coupled services. The services in SOA are independent, on demand, and can be accessed without knowledge of how they are implemented. Reference [8] discusses in detail these three architectural paradigms, and the standards implementing them, in the context of developing software for robotics. The paper also compares these paradigms with respect to various factors which include granularity, coupling, reusability and extensibility.

CORBA [9] is a standard defined by the Object Management Group (http://www.omg.org/), and implements DOA. It enables software components that are written in multiple programming languages and running on multiple computers to work together. CORBA achieves this by using an interface definition language (IDL) to specify the interfaces that objects expose to the outer world. CORBA then specifies a mapping from IDL to a specific implementation language like Java or C++. There are standard mappings for many common languages including: Java, C++, C and Python. Non-standard mappings also exist for languages including Perl and Visual Basic, and are implemented by writing object request brokers (ORBs) for them. COM [10] is a family of standards (one example being the well-known 'ActiveX') very similar to CORBA and was introduced by Microsoft for inter-process communication and also for dynamic creation of objects in a wide range of programming languages. Distributed Component Model (DCOM) is a technology that was introduced as part of the COM family for communication
between components that are distributed over a network of computers. It was a competitor to CORBA and has an architecture that is very similar to that of CORBA.

Jini also called Apache River [11] is an architecture that implements SOA and is often used in dynamic computing environments to build network systems that are scalable, flexible and evolvable. Jini has three main components which are the client, server, and lookup service. The service is a resource which needs to be made available in the distributed environment, the resource can be a physical device (like a disk drive or a printer) or it can also be a software service (like an authentication service or a database service). All services register themselves to the lookup service so that they can be found and used. The client is any user of a service and contacts the lookup service in order to find the service. Reference [12] discusses at length about Jini and also about other architectures, open source tools and frameworks that can be used to develop laboratory automation software as per the principles of SOA.

An application programming interface (API) for inter-process communication called .NET Remoting [13] was released by Microsoft as part of the .NET 1.0 version. In NET Remoting, a remotable object is one that can be accessed by client applications that are on a different computer. The remotable object is accessed by clients using a proxy which implements the same interface as that of the remotable object, however, the proxy does not have any actual implementation, and delegates all the calls to the remote object. The proxy makes use of a channel to communicate to the remote object, and through the proxy, the client instantiates and uses a remotable object as if it were a local object. Windows Communication Foundation (WCF)
[14] is a .NET an API introduced by Microsoft that superseded .NET Remoting. WCF is used to build distributed service oriented applications. WCF services are loosely coupled to each other and can be consumed by multiple remote clients. A WCF client can also consume multiple WCF services. A WCF client communicates with a WCF service through an “endpoint” of the service. Each endpoint consists of a contract that specifies the operations exposed by the service, an address where it can be found, and a binding that specifies what communication protocols have to be used to communicate with the service.

The API called OpenMP [15] supports shared-memory parallel programming and supports most processor architectures and operating systems. In the OpenMP programming model, there is a master thread that spawns a specified number of slave threads and work is divided equally among them. The threads are allocated to different processors and operate concurrently. Message Passing Interface (MPI) [16] is a message passing system for developing large scale systems that are portable and scalable. Task Parallel Library (TPL) [17] is a set of APIs provided by Microsoft as part of .NET 4.0 and simplifies the process of adding concurrency and parallelism to applications. TPL takes care of many low-level details, which includes partitioning of work, cancellation support, scheduling of threads and state management thereby freeing the developer from being concerned with such issues. The programming language called Erlang [18] was developed by the telecommunications company Ericsson to be used to develop scalable, distributed, robust, real-time applications. It is a concurrent programming language, and runtime system, that supports the capability of changing code without stopping the system (also known as hot swapping). Erlang processes communicate with each other using message passing instead of shared
memory and hence, they do not use locks. Applications developed using Erlang are mainly used in the domains of banking, telecommunications, computer telephony, e-commerce and instant messaging.

A large amount of commercial software is currently available for various types of laboratory automation systems. Some of the manufacturers of such software are Thermo Scientific, Tecan, Biocompare, Hudson Robotics, Beckman Coulter, PerkinElmer, Process Analysis & Automation and Tessella. These systems satisfy many of the current research needs, which include software for automating standard tasks, high precision imaging and microscopy, automated microplate loading, and liquid handling. The current state of the art in high throughput automation systems has been given in [3]. Although these systems take care of many of the current research needs, the proprietary nature of their hardware and software architectures make them very inflexible and nearly impossible to be modified; thus, rendering them useless for custom software development, or modification to suit research specific needs. Added to that, the amount of money that is required to procure most commercial systems is often time prohibitively expensive. Many laboratories exploring new research or findings are unwilling to invest in, or unable to afford, these products.

In addition to commercial laboratory automation software, there are many frameworks used for developing robotics software, many of which are open source; these are relevant here because software that is developed for robotics applications share many of the traits of, as well as solve many of the problems associated with, laboratory automation software. Miro [7] is a CORBA (Common Object Request Broker Architecture) based distributed object oriented framework for mobile robot control. The core components of
Miro are developed in C++ and are targeted for the Linux operating system. However, since CORBA is programming language independent, components of Miro can be written in any language and for any platform that supports CORBA. Orca [8] is an open-source framework for developing component-based robotics software. It provides a means for defining and developing the building-blocks which can be combined together to build robotic systems of arbitrary complexity, ranging from single vehicles to distributed sensor networks. RT-Middleware (RTM) [9] is a software platform that combines the software modules of the robot functional elements (RT functional element) to construct a robot system (RT system). An RT functional element is a robot component that provides certain functionality. A RT functional element can be related to hardware such as a device unit like Sensor, Camera and Servomotor or a combination of device units such as Mobile platform and Arm, or it can also be related to software such as control algorithm or image processing algorithm.

While not open source, Microsoft Robotics Developer Studio (MSRDS) [19] is a .NET Software Development Kit (SDK), provided by Microsoft, for developing robotics software. MSRDS contains two toolkits that greatly enhance, as well as simplify, the development of concurrent and scalable applications: the Concurrency Co-ordination Runtime (CCR), and Decentralized Software Services (DSS). CCR [20] is a .NET library developed to simplify the implementation of thread-safe asynchronous behavior in software applications. One of its most important additional features is: software that has been developed using it will scale seamlessly with the number of processing units that the software is used with. CCR has powerful coordination primitives that greatly simplify coordination between
asynchronous tasks; it also handles many low-level details, which includes thread creation, deletion and scheduling. CCR also has failure handling mechanisms that handle failures spanning multiple threads of execution (this includes partial failures) in a robust manner. DSS [21] is a .NET library used to write service oriented, scalable, distributed applications, and it makes extensive use of the CCR. DSS services are capable of being distributed across multiple computers on a network, and use message passing via the Decentralized Software Service Protocol (DSSP), to communicate with each other. DSSP is a SOAP like protocol, and separates the state of a service from its behavior. DSSP has the added attribute of providing asynchronous notification of events, another benefit that fosters true asynchronous software development. With the help of CCR, DSS supports error handling on software that is distributed across multiple computers; this is a notable advantage over other available tool-kits. The problems that CCR and DSS solve, along with the numerous benefits that they bring to developing asynchronous distributed and scalable software, have led to them even being used in many non-robotics applications [22].

Another framework worth mentioning here, due to its popularity and widespread use in laboratories requiring microscopy automation, despite its limited application domain, is μManager [24]. μManager is an open source and freely available software package for automating microscope operations. μManager works with the microscopes manufactured by major manufacturers such as Leica, Nikon, Olympus and Zeiss, most scientific-grade cameras, and many microscope imaging peripherals [25]. μManager provides a comprehensive, freely available, imaging solution; including a user interface with which the user can automate common microscope image acquisition
operations such as multi-channel imaging, and time-lapse and z-stack acquisitions. μManager can be fully integrated with ImageJ [26], a widely used image processing software package developed by the National Institutes of Health (NIH). ImageJ is Java based multithreaded software, and can be run either as an applet or as a downloadable application. It can display, analyze, process, save and print: 8 bit, 16 bit and 32 bit images, as well as read many image formats such as: BMP, DICOM, GIF, JPEG, and TIFF. It supports standard image processing functions and geometric transformations.

The Interchangeable Virtual Instruments (IVI) [27] standard defines an open instrument driver architecture that abstracts out the common functionality inherent in various types and classes of instruments. The standard’s goal is to simplify the interchangeability of instruments, provide improved performance, and reduce the cost of program development. IVI also provides a basic framework consisting of a set of formally defined abstract instrument classes, as well as shared software components. IVI is mainly utilized in the development of software for automating the use of test and measurement equipment.

Although all of these existing architectures, standards, APIs, programming languages and frameworks solve numerous problems, they are targeted to a much wider, more diverse, domain of application software development. In order to be utilized effectively to solve the problem at hand, they need to be customized and combined in various, non-trivial ways, to develop software systems that deal with many of the nuances of device control, as well as the problems of thread-safety and responsiveness.
**CONTRIBUTIONS OF THE THESIS**

Component Oriented Distributed Architecture (CODA) has been developed. The architecture is based on the concepts of reusable components that can be distributed across multiple computers, and used in concert to develop lab automation software that is distributed, thread-safe and responsive. CODA Framework has also been developed based on the concepts of the architecture and is used for developing software that implements CODA. CODA Framework provides the requisite infrastructure that a developer can use as a starting point, to build a robust, extensible, distributed automation system.

The core of the CODA Framework is a set of .NET base classes that provide the basis for developing reusable asynchronous components that implement consistent state machine architecture. The framework ensures the state machine operates in a thread safe manner, as well as providing an interface decoupling mechanism that simplifies distributing, and using, the components across remote computers. The framework also provides a set of components that provide software abstractions for some of the more common hardware devices found in laboratory automation systems. These components allow the developer’s software to be extensible, since the components greatly simplify the integration of new hardware in to the software (provided an abstract component exists for the type of device the developer wishes to add).

The framework heavily leverages off of the power provided by the Microsoft Robotics Developer Studio (MSRDS): it makes use of the DSS toolkit to provide the asynchronous distributed computing capability required by the component framework; it also makes extensive use of the CCR toolkit as the
means for providing thread safe, asynchronous behavior, and inter-component message passing.
CHAPTER 2
CODA

This chapter explains Component Oriented Distributed Architecture (CODA) that has been developed as part of the thesis. The architecture is based on components that could be distributed across multiple process spaces and computers. The chapter explains the key concepts and ideas of the architecture and also describes how the architecture solves some of the key challenges of developing laboratory automation software namely responsiveness and thread-safety.

BASIC CONCEPTS AND DEFINITIONS

This chapter explains the basic concepts and definitions used by CODA and are essential for understanding CODA. Some of these concepts and definitions are based on the concepts and definitions used in component oriented architectures, asynchronous message passing and agent based programming.

- **Component**: Independent, autonomous software module that performs a useful task, or related set of tasks. A component realizes its autonomous nature by executing its requested behavior in an asynchronous manner (non-blocking operation), and limiting it’s interaction with the outside world to a predetermined set of messages that are sent to its main messaging port (it’s API), or a notification port.

- The outside world, i.e. the user of a Component, requests the Component to perform a particular behavior (operation) by posting (i.e. sending) the appropriate API message to the Component’s main message port.
• A Component supports the concept of state, which is its current condition, or ‘context’; and is determined by the value of its internal data, as well as what operation, if any, it’s currently executing.

• Upon receipt of a message at the Component’s main message port, if the message is valid for the Component’s current state, the Component will perform the requested operation.

• A Component supports the concept of notification, whereby changes of state, as well as component specific events, trigger the posting of notification messages to a user supplied notification port.

• State Data: The entire aggregate of a Component’s internal data.

• State: A Component’s current condition or context of operation, as determined by its State Data combined with any operation/s it is currently in the process of executing.

• Property: An individual element of the Component’s State Data; used to affect the Component’s behavior, or store information about the Component or its current condition.

• Operation: Behavior that is executed by the Component in response to a Message.

• Message: The means by which the outside world can interact with a Component. A Message is a software construct that provides message identity (i.e. what the message is), as well as the data required to process the message, i.e. the data required to perform the requested Operation. Component Messages can be divided into the following broad categories:
  • Command: General request to perform a particular task or operation.
• Set: Request to set a particular property to a desired value.

• Data Request: Request to retrieve the current value of a Component’s internal data.

• Event: A message which indicates that an external event, that may affect the component, has occurred.

• Port: A software construct that provides a mechanism for receiving Messages and transferring them to another module for further processing.

• API: Predefined set of Messages that define, and are used to execute, the Operations that the user of a Component may request. This set of messages defines the ‘interface contract’ that the component has with the outside world.

• Main Message Port: Means by which the outside world may send messages to a Component. API Messages are posted to this port to request that the Component perform a particular Operation.

• Notification: Mechanism by which changes of state, as well as the occurrence of component specific events, are communicated to the outside world. These ‘events’ are communicated to a user of the Component by posting the appropriate Message to a user supplied notification port.

• Notification Port: Means by which the Component communicates with the outside world. The Component posts Notification Messages to this Port to notify the user of the Component that a particular event or change of State has occurred. Notification Ports are passed to the Component via API Messages utilizing the Main Message Port.
• Types of Notifications:

Notifications can be divided into two broad categories:

• Static Notifications: The class of Notifications where State change and Component event Messages are posted to a user supplied Notification Port. The Notification Port is supplied to the Component via an API “subscribe” message; this Port is then stored internally by the component for use in all Static Notifications.

• Dynamic Notifications: The class of Notifications where The Notification Port is passed along with the rest of the data in a normal API Message, and is used to notify the user of Message/Operation specific events - specifically: Message Acknowledgement Notifications and Operation Complete Notifications. The Notification Port is stored temporarily by the component, for the duration of the requested Operation.

• Acknowledgement Notifications (ACK): A type of Notification that is used to send an acknowledgement Message to the user of a Component; indicating that the desired Message has been received by the Component, and that it is valid.

• Operation Complete Notifications (OPC): A type of Notification that is used to let the user of the Component know that the requested Operation has completed its execution.
Figure 1 below explains the high level design of a CODA component.

**Figure 1. Component High Level Design**

The constituents of the high level design of the component are explained in detail below.

- **User Layer:** Software that sends messages to a Component’s Main Message Port, and receives messages from a Component’s Notification Port.

- **Component Infrastructure Layer:** Component infrastructure software that is used to process incoming Messages from the Main Message Port, as well as process Notification requests from the Component implementation.

- **Exclusive Message Processing Dispatcher:** Receives and processes Messages from the Main Message Port, and then dispatches Operation requests to the Implementation Layer.
• Notification Mechanism: Receives Notification requests from the Implementation Layer and posts the appropriate Message to the Component’s Notification ports.

• Component Implementation Layer: Composed of implementation of Component specific behavior and Component State Data. Used to perform requested Component Operations; and when required, send Notification requests to the Notification Mechanism. The architecture allows the Component implementation to post Messages from within the Component Implementation Layer to the Component’s Main Message Port to invoke Operations, as it sees fit. These Messages are known as Internal Messages. Posting Internal Messages that update State Data from within asynchronous Component implementation code (e.g. Task implementation code) ensures thread safe operation of the Component.

• Internal Messages: Messages sent from the Implementation layer to the Component Infrastructure layer to be processed by Exclusive Message Processing Dispatcher.

As stated in the basic concepts and terms, a CODA Component is used by making a request that invokes some behavior, formally known as an Operation. From the perspective of a Component user, i.e. a person who will make use of a CODA Component in their software (as opposed to a developer who will be implementing and developing their own CODA component, that conforms to the defined architecture), Component usage is quite straightforward, provided the user is aware that the Component Operations are always performed asynchronously with respect to the thread
of operation that invoked the operation: i.e.-when a Component Operation is invoked, the software that invoked the Operation does not stop and wait for it to complete, it continues on with its own execution, in “parallel” with the Components Operation. The user will be sent a Notification when the Operation completes (OPC Message).

Component Operations are divided into two broad categories: Actions and Tasks. From a strictly user perspective, Actions can be thought of as simple or trivial operations that take a short time to execute (e.g. < 100 ms), and are not prone to error. An example of an Action would be an Operation that does some minor manipulation or processing of Component data. An important consideration when invoking Component Actions that a user needs to be aware of is: Component API messages will not be processed while an Action is executing; i.e. if the Component receives a Message to perform a new Operation while an Action is executing, the Message will not be processed until the current Action is finished; hence, the new Operation will not start until the current Action is done. This ensures that all the actions are executed in a thread safe fashion. The updating of state data can only be done in an Action and hence the state data of the component is thread-safe with respect to other actions and also tasks. The way an action is executed is explained below in Figure 2.
From a user perspective, Tasks are Operations that perform some complex, or lengthy task, that could be prone to error. An example of a Task would be an Operation that communicates with a database, or device (hardware); or execution of a lengthy complex algorithm that may experience an error. Unlike Actions, Tasks do not ‘block’ Message processing; i.e. if the Component receives a Message to perform a new Operation while a Task is executing, the Message will be processed, and the new Operation will execute while the Task is executing, concurrently. This is a very powerful feature that allows the Component to be accessed while lengthy or complex Operations are executing (if it’s within the semantics of the specific Component type, i.e. if it’s ‘allowed’) thereby enabling the component to be responsive. A Task is not allowed to update the state data directly and can only do so by posting a message and causing an Action to execute. This
allows Tasks to update state data in a thread-safe fashion. The way a task is executed is explained in Figure 3 below.

**Figure 3. Task Execution**

**REQUIRED BEHAVIOR, OPERATIONS AND PROPERTIES**

An integral part of the CODA Component architecture, every component must implement certain required behaviors, operations and properties which complete a design pattern that ensures the goals of the CODA architectural design are met. This amounts to an ‘inversion of control’
that enforces a consistent reliable behavioral pattern that both simplifies, and adds robustness to CODA component usage. The required behavior, operations and properties that are required by every component in the framework is discussed in detail below.

**Required Behavior**

After a Component has been created, the following sequence of behaviors must be accomplished before the Component can be used: First the Component must be configured, and then it must be ‘opened’. Once the Component has been successfully opened it is ready for use. After the user of a Component is through using it, the Component must be ‘closed’. Only after the Component has been closed, may it may be destroyed. This is discussed in detail below.

- After a Component has been created, the Component must be configured using one of the required family of configuration Operations (defined below).

- Configuration: The act of setting state data to values required to successfully open the component; as well as setting properties to desired initial values. Successful Configuration must be achieved before the Component can be Opened.

  - Configuration Data:

    - Values for state data that are required to successfully open a Component: e.g. comm port and baud rate for a Component that utilizes serial port communications; “connection string” for a Component that utilizes an SQL database; or device ID and channel number for a data acquisition component.
• Desired initial values for state data (e.g. properties) to be used after the Component has been successfully opened.

• Family of Configuration Operations:
  • Configure: The act of configuring the component using Configuration Data that is passed to the Component as part of the Configure Message.
  • SetDefaultConfiguration: A configuration Operation that sets the required state data to default values that are determined by the design of the Component.
  • LoadConfigurationFromFile: A configuration Operation that sets the required state data using Configuration Data that is loaded from a file that is specified in the Message data.

• Opening: The act of setting up the Component for use. Any initialization, startup procedures, or resource allocation that the Component may require should be done in the Open Operation. Some examples of typical procedures that are performed during the Component opening process include: opening the comm port for a Component that utilizes serial port communications; connecting to a database.

• Closing: The act of ‘cleaning up’ a Component after use. Any clean up, shut down procedures, or resource de-allocation, that the Component may require should be done in the Close Operation. Some examples of typical procedures that are performed during the Component closing process include: closing the communication port for a Component that utilizes serial port communications; disconnecting from a database for
a Component that utilizes an SQL database; or closing a session to a data acquisition device, for a data acquisition Component.

Required behavior illustrated as a Finite State Machine (FSM): A classic FSM state chart provides a succinct and easy to understand way of illustrating the architectural behavioral requirements for a CODA Component. It is important to note that while a state chart provides an ideal way of completely elucidating the required behavior for a Component, there is no requirement to use an actual state machine in the implementation of a Component. The following state chart in Figure 4 illustrates the semantics and behavior that are required for a CODA Component.

![Figure 4. Component Behavior FSM](image)

In addition to the behavior and semantics from the previous section, there are other Operations and Properties that comprise the required
Component infrastructure, which complete the desired architectural design pattern. The required Operations and Properties for a Component that complies with the CODA architecture are listed below.

**Required Operations**

- **Configure**: Explained in the previous “Configuration” section.
- **SetDefaultConfiguration**: Explained in the previous “Configuration” section.
- **LoadConfiguration**: Explained in the previous “Configuration” section.
- **Open**: Operation that sets up the Component for use. See the previous “Opening” section.
- **Close**: Operation that cleans up the Component after use. See the previous “Closing” section.
- **ClearError**: Operation that clears any error information that has been cached in the Component as the result of a previous error, and sets the ErrorOccurred Property back to <False>. Invoking this Operation implies that the user is aware of the error, and that any desired error handling has been accomplished.

**Required Properties**

- **Name**: Read only property that is used to identify the particular instance of a Component.
- **CreationTime**: Read only property that is used to identify the time when the Component was created.
- **Errored**: Read only Boolean property that is used to indicate that an error has occurred during a Component Operation. The semantics for this property are as follows: Once an error has occurred during an Operation, this property is set to <True> indicating to the user that an
error has occurred. At any time after this, the user may invoke the ClearError Operation to acknowledge the error; implying that the user is aware of the error, and that any desired error handling has been accomplished.

- Exception: A software construct that is used to store relevant information for the most recent error that has occurred with the Component.
- Opened: Read only Boolean property that is used to indicate that the component has been successfully Opened. This Property must be set to <False> upon Component creation, and then set to <True> after the Component has been successfully Opened. The Component must set this Property back to <False> immediately after the Component has been Closed.

**COMPONENT USE AND LIFETIME MANAGEMENT**

In addition to the basic Component architecture as previously described, further ancillary aspects of the CODA Component architecture define the process of how a Component is used, and include: the concept of an interface object, which is used for all interactions with a component; architectural directives dealing with component lifetime management (i.e. Component creation and destruction).

**Basic Concepts and Definitions for Component Use**

The software that makes use of a Component is referred to as Client software, or ‘the client’. In order to make use of a Component, the Client software must first create it, after which, Operation requests may be sent to the Component to accomplish the desired functionality the Component implements. The Client software may simply be part of an application that
uses the Component, or it could be another Component (Collaboration, which is explained in a later section).

The CODA architecture also supports the concept of remote Component use, which is where the Client software resides in a different executable program than the executable program where the Component resides (i.e. different process space, including possibly, a different networked computer) – this is referred to as Remote Use; when the Component is not used in this way, and instead, the Client software resides in the same executable program as the Component, the Component is said to be in Local Use.

The executable program where a Component resides is referred to as the Host Software for that Component. In the case of Local Use, the Client software and Component reside in the same executable program, hence the Client software is the Host software for the Component. In the case of Remote Use, the Client software and Component reside in different executable programs, perhaps on different networked computers; in this case, the Client software and Host software are completely separate entities. All of these terms are defined formally below.

- Host software: This is the executable program where a Component resides. For a simple application that makes Local Use of a Component, the application itself is the Component’s Host software. In the case where an application makes Remote Use of a Component, the Component’s Host software is a separate program from the application program. In fact, the Host Software could be a program running on a remote networked computer, which was
written for the express purpose of hosting the Component remotely, and making it available to Client software.

- **Client software**: This is the software that actually makes use of a Component, i.e. - from the perspective of a particular Component, it is the ‘user’ of that Component. In order to make use of a particular type of Component, the Client software must first create the desired Component, the Component can either be created locally, or remotely (explained in a later section). Once the Component has been successfully created, the Client software may then make use of the Component by sending it requests to perform the desired operations. After the Client software is finished using the Component, it is its responsibility to destroy the Component.

- **Local Use**: The scenario where the Client software resides in the same executable program as the Component of interest, i.e., they reside in the same process space. In this case, the Component’s Host software is the same program as the application running the Client software, or put another way: the Client software application hosts the Component.

- **Remote Use**: The scenario where the Client software resides in a different executable program than where the Component of interest resides, i.e. they reside in different process spaces. In this case, the Component’s Host software and the program running the Client software are two entirely different programs, which could possibly be on different computers that are on the same network.
**Interface Object**

The Interface Object is a software construct that is used to simplify the process of using a Component from Client software. The Interface Object for a particular Component is obtained from the process that is used to create the desired Component (explained in the next section); it simplifies Component use by providing a mechanism that handles many of the repetitive and tedious steps Client software would have to go through to effectively use a Component. In addition, the Interface Object also provides an abstraction that allows the Client software to interface with a Remote Use Component the same way it does with a Local Use Component. Since, from the perspective of the Client software, using a Component is the same regardless of whether it’s a Local Use or Remote Use Component, switching between using a Remote or Local version of a particular Component is seamless and encourages proper system design. The following are the advantages of the interface object.

The Interface Object acts as a ‘wrapper’ around the Component (or Remote Transport Mechanism) that shields the Client software from performing tedious and repetitive tasks needed to use a Component, such as: creating messages and populating them with data (i.e. the data required for the requested Operation - the ‘arguments’); creating Notification Ports for receiving notifications from the Component; waiting for an ACK response from the Component, etc.

The Interface Object implements the following features to simplify Component use.

- ‘Wraps’ the process of requesting that a Component execute a desired Operation by exposing a routine that can be executed as a simple function call.
• Data required for a desired Operation is passed as normal arguments to the appropriate ‘wrapping routine’ function call.

• Creates and populates the appropriate Component Message that will be posted to the Component to execute the desired Operation.

• Posts the Message requesting that the Component execute the desired Operation, and then waits for the ACK Notification from the Component.

• Returns, from the appropriate ‘wrapping routine’, the Operation’s OPC Port, which the Client software can then use as it sees fit.

• Provides a routine for destroying the Component.

• Remote Component Use: The Interface Object abstracts out the process of using a Remote Use Component. In addition to shielding the Client software from the complexity of interacting with a Remote Component, the fact that the Client software interacts with the same interface, whether Remote or Local, means that a particular Component being used by the Client can be switched from Local Use to Remote Use, with little to no change in the Client software. This can be a huge advantage when upgrading or modifying the design of a software system.

  • From the perspective of the Client software, the use of a Remote Use Component, via the interface Object, is identical to using the same Component as a Local Use Component.

  The diagram in Figure 5 below illustrates how the Interface Object works, as well as how it is used by Client software.
**Component Lifetime Management**

There needs to be a Component Management Library which is nothing but a software library that provides routines for lifetime management of CODA Components. The library will contain routines that can be used to create a particular type of Component, and its accompanying Interface object; as well as routines that can be used to properly destroy a particular type of Component, along with its accompanying Interface object.

Components can be created either for local use or remote use. In the case of a local use component, the component is hosted by the Client software’s application program. To create the Component, the appropriate

---

**Figure 5. How the Interface Object Works**
routine in the Component Management Library is run. The only required data for the routine is the name the Component will have. Once the Component has been successfully created, the creation routine will create an Interface Object for the Component, ‘Connect’ the Component to it, and return it to the Client software.

In the case of a remote use component, the Host software is different than the Client software’s application program. To create the Component, the appropriate routine in the Component Management Library is run. From the perspective of the Client software, except for supplying additional location information about where the Component’s Host software is located, the process of creating a Remote Use Component is the same as creating a Local Use Component.

The moment the client is done with using the component, it needs to properly destroy the component. A Component can only be destroyed after it has been Closed. To destroy the Component, the Client software need only call the Interface Object’s Destroy routine. The process is the same for Local Use, or Remote use Components; the added intricacies of destroying a Remote Use Component are handled internally by the Interface Object.
CHAPTER 3
CODA FRAMEWORK

As part of the thesis, a CODA Framework has also been developed that implements the ideas proposed in CODA. The framework has been designed based on UML state machines, Concurrency Coordination Runtime (CCR) and Decentralized Software Services (DSS) and is used to develop software that is distributed, scalable, thread-safe and responsive. The framework has a base component from which all the other components of the framework derive from. The framework also has components to control cameras, mechanical stages and also to do imaging. This chapter discusses the concepts used by the framework to implement the ideas of the architecture. This chapter also discusses how components are designed in the framework, the mechanism involved in creation and destruction of components and also in client-component communication, and the device abstractions that have been developed by the framework for cameras and stage controllers. For more details on the implementation at the code level, [28] should be referred.

CONCEPTS USED

As mentioned earlier, the framework makes use of UML state machines [23], Microsoft Robotics Studio (MSRDS) libraries CCR and DSS to implement the ideas that has been proposed by the architecture. A good understanding of them is very important to understand the framework design and implementation. In this section an overview of these concepts and how they have been used to implement the ideas of CODA has been discussed.

UML State Machines

UML state machines [23] are used widely to model the behavior of a system. UML state machines are broadly classified into two types – behavioral
state machines and protocol state machines. Behavioral state machines model
the behavior of a system as a set of states and transitions. Protocol state
machines do no model behavior but they rather model protocols. The
framework makes use of behavioral state machines. Some of the main
concepts of behavioral state machines have been outlined below

- State: A state of a system is a current condition, or 'context'; and is
determined by the value of its internal data, as well as what operation,
if any, it's currently executing. In UML state diagrams, a state is
represented as a rounded rectangle. A state can have three kinds of
behavior – entry, do and exit. Entry behavior is executed as soon as a
state is active, do behavior is executed while the state is active and
exit behavior is executed just before the state becomes inactive. The
internal behaviors of a UML state machine is represented as
label/behavior.

- Transition: A transition is a change of state from a source state to
target state in response to an event (also called trigger). Transition is
represented as an arrow emanating from the source state and pointing
at the target state. A transition can optionally have a guard condition.
A guard condition is a Boolean condition and is associated with a
transition and decides whether a transition should fire or not.
Transition behavior: A transition also has a description that is written
on top of the transition arrow that describes the circumstances under
which the transition occurs. The complete representation of a
transition description is trigger[guard]/behavior in which each of the
constituent elements are optional. Internal Transitions are special kind
of transitions but they do not change the state of the system and hence they do not have a target state.

- Extended State: In many scenarios, merely having a single state is not sufficient to represent the entire state of the system. Such a design could lead to proliferation of the number of states. In such a scenario, the state of the system is made up of a primary state and a set of extended state variables. The primary state together with the extended state variables is called an extended state.

UML State machines are highly powerful tools to model reactive systems. Reactive systems are those who response to events is dependent on the context or the state at which the system is currently in. Since, almost all devices and in particular laboratory devices, have a requirement that it can only do certain set of operations in a certain context or state, the code that controls the devices can be modeled effectively by the use of UML state machines. One of the other advantages of UML state machines is that they are deterministic. If the same sequence of events is sent to the state machine any number of times, it produces the same result.

The major part of the implementation layer (the one that deals with executing Action and Task operations) of the components in the framework is implemented by using UML state machines. Actions are implemented as UML transition behaviors and tasks are implemented as UML state behaviors. The only way by which a component can modify state data is by posting a message and causing a transition. This ensures that the state data is thread-safe. The state behaviors are executed asynchronously and this ensures that the component is responsive while a state behavior is executing. The component has a state machine engine that is responsible for the processing
of messages. Note that the state machine engine is only responsible for the processing of trigger messages (trigger messages are messages that modify the state data) and the non-trigger messages (messages that do not modify the state data) are processed outside of the state machine engine. Since most of the messages that are posted to the component falls are trigger messages, the state machine engine is responsible for the processing of most of the messages.

**Microsoft Robotics Studio**

Microsoft Robotics Developer Studio (MSRDS) [14] is a .NET Software Development Kit (SDK), provided by Microsoft, for developing robotics software. MSRDS contains two toolkits that greatly enhance, as well as simplify, the development of concurrent and scalable applications: the Concurrency Co-ordination Runtime (CCR), and Decentralized Software Services (DSS). CCR [15] is a .NET library developed to simplify the implementation of thread-safe asynchronous behavior in software applications. One of its most important additional features is: software that has been developed using it will scale seamlessly with the number of processing units that the software is used with. CCR has powerful coordination primitives that greatly simplify coordination between asynchronous tasks; it also handles many low-level details, which includes thread creation, deletion and scheduling. This section covers only a brief discussion of the core concepts of Concurrency Coordination Runtime and Decentralized Software Services. It is highly recommended for the framework users to refer to the official documentation of Microsoft Robotics Studio for a more detailed explanation of all these concepts.
CCR is a .NET library that helps applications exploit concurrency and also provides a very elegant way to deal with partial failures. The core primitives of the CCR library are discussed below.

- **Tasks**: A task is nothing but a work item that can be executed by a thread. All the tasks in CCR implement the ITask interface. This provides a very easy and elegant abstraction for work items to be passed around and queued to be executed.

- **Dispatcher**: Dispatcher is nothing but a thread pool that has a number of threads. CCR tasks are executed by the threads in a dispatcher. The priority of the threads in the dispatcher could be set to different levels such as low, medium, high etc.

- **Dispatcher Queue**: Dispatcher Queue is a FIFO data structure where CCR tasks get queued. A dispatcher is attached to Dispatcher Queue to execute the tasks in the dispatcher queue. A dispatcher can execute tasks from multiple dispatcher queues whereas a dispatcher queue can only have one dispatcher to have its tasks executed.

- **Port**: Ports are queues that can receive messages of the same type with which the port was created. This ensures type safety at compile time and prevents messages of other types from being queued. To actually place a message in a Port, the user of the Port “posts” the message. A message remains in the Port until it is picked up a Receiver. The execution of the receiver happens asynchronous to the thread that is posting the message.

- **PortSet**: PortSet is a collection of Ports. You can individually access each of the Ports in the PortSet and attach receivers to them.
• Arbiter: CCR implements coordination primitives through arbiters to synchronize tasks. An arbiter is also a task and all arbiters derive from ITask. The most commonly used arbiters are discussed below.

  • Receiver: Receiver arbiter is the most basic arbiter and is attached to a Port. The receiver gets fired whenever a message arrives at the port.

  • Choice: The choice arbiter is equivalent to an XOR on two receivers and waits on two receivers until one of them fires. It then shuts down the other unused receiver.

  • Join: The join arbiter is equivalent to an AND on two receivers and waits for the two receivers to complete before proceeding further in execution.

  • Interleave: Interleave arbiter is comparable to reader-writer lock style synchronization. Interleave arbiter has three groups of receivers namely exclusive receiver group, concurrent receiver group and teardown receiver group. A receiver from the exclusive group runs exclusive to other receivers in its group and also exclusive to receivers in the concurrent and teardown group. A receiver from the concurrent group runs concurrent with respect to the other receivers in its group. A receiver from the tear down group runs exclusive to other receivers from its group and also exclusive to receivers in the concurrent and exclusive group. Once a receiver from the tear-down group is done processing the message, it shuts down all the receivers in the Interleave.
DSS [16] is a .NET library used to write service oriented, scalable, distributed applications, and it makes extensive use of the CCR. DSS services are capable of being distributed across multiple computers on a network, and use message passing via the Decentralized Software Service Protocol (DSSP), to communicate with each other. DSSP is a SOAP like protocol, and separates the state of a service from its behavior. DSSP has the added attribute of providing asynchronous notification of events, another benefit that fosters true asynchronous software development. With the help of CCR, DSS supports error handling on software that is distributed across multiple computers; this is a notable advantage over other available tool-kits. Every DSS service has a state that has service’s current condition or context of operation. A DSS service also has an operations port which is nothing but a CCR PortSet that is used by the users of the DSS service to communicate with the service. The users of the DSS service communicate with it by posting messages to its operations port. Every message in DSS has a body and also has a response port which is nothing but a CCR PortSet.

The following are the mapping of the ideas of the architecture to how they have been implemented in the framework using CCR.

- **Messages:** Both the main messages and internal messages are C# generic classes that have a BODY that can of any type, an acknowledgement port which is nothing but a CCR Port and a completion port which is again a CCR Port.

- **Messaging Port:** The main messages port of a component is a CCR PortSet. The users of the component communicate with the component by posting messages to it. In addition to the main messages port, the
component also has an internal messages port, which is also a PortSet, to which the component posts internal messages.

**COMPONENT DESIGN**

The design followed by any component in the framework is given in Figure 6 below.

![Diagram of Component Design](image)

**Figure 6. Component Design**

The idea of a transport has not been discussed in the architecture and is been introduced by the framework. The transport abstracts out all the communication details that are involved in communicating with a component. The interface object communicates with the component by making use of the transport. The framework supports two transport mechanisms, and they are local and DSS. The local transport mechanism is used when you want to create and use a component in the same process space. The DSS transport mechanism is used when you want to create and use a component in a different process space either in the same computer or across different computers.
At the heart of the design is the ICodaComponent. Any component in the framework needs to implement the interface. The Component here refers to any component in the framework. There is an ITransport interface that derives from the ICodaComponent and that is the interface that needs to be used when posting messages to the component. The DssClient represents the DSS transport mechanism and the LocalTransport class represents the local transport mechanism. The InterfaceObject is the interface object for the Component and contains all the wrapper methods for the messages supported by the Component. The InterfaceObject talks to the Component by making use of the ITransport interface. So, the interface object is abstracted out from the transport mechanism used to communicate with the component.

**BASE COMPONENT**

As mentioned in the architecture, there are some required behavior, operations and properties for every component and it could be abstracted out in a base component. CodaComponent is an abstract class that acts as a base class for all the other components of the framework. This class abstracts out the state data, the main messages, the internal messages port, the state and the state data that every component implementing the CODA architecture must have. It also has the structure for the required behavior for all CODA components with abstract methods for their implementations that the derived components can implement. The base component also takes care of the internal details of creating and disposing the CCR infrastructure. Figure 7 below describes the design of the CodaComponent.
CodaComponent has the following constituents.

- **CodaComponentStateData**: The minimum state data that every component should have.
- **CodaComponentState**: An enumeration representing the states of the base component (Shutdown-Configured-Ready).
- **CodaComponentMainMessagesPort**: This class derives from a CCR PortSet and contains all the API messages that are posted to the component.
- **CodaComponentInternalMessagesPort**: This class derives from a CCR PortSet and contains the internal messages that are posted from within the component.

- **StateMachineEngine**: This class contains the state machine engine logic that is used to process the trigger messages that are posted to the component.

Just like a base component, the framework also has a base interface object, a base DSS client and also a base DSS service. The base interface object has wrapper methods for all the API messages exposed by the base component. Derived components can also derive from the base interface object and add API methods that are specific to the component. The base interface object is also responsible for the creation and destruction of CCR resources that is needed to communicate with the transport. The base DSS client and base DSS service provides the entire DSS infrastructure that is needed to communicate with components that are remote. The user only needs to derive them and add component specific messages and the framework takes care of passing the messages using DSS, and also for creating and destroying the DSS infrastructure. A detailed explanation of how the user can use the framework to create components both for local and remote use is explained in Appendix A.

**COMPONENT LIFECYCLE**

The client software needs to create the component before it can use. Once the client software is done using the component, the component needs to be destroyed. The steps involved in the creation and destruction of a component is different depending on whether the component is located local or remote and has been defined in the framework design. This ensures that
the component can be created and safely disposed in a very consistent manner and is also very helpful in troubleshooting. Figure 8 illustrates the creation and destruction of a local component. It could be seen that the local transport mechanism is used for local components.

**Figure 8. Local Component Lifecycle**

Figure 9 illustrates the creation and destruction of a component that is remote. The transport mechanism used in this case is DSS.
Figure 9. DSS Component Lifecycle

CLIENT-COMPONENT COMMUNICATION

The client does not communicate with the component directly but does so by making use of the wrapper methods provided by the interface object. The interface object forwards the message to the transport mechanism, which in turn is responsible for forwarding the message to the component. The steps involved in client-component communication for both local and remote has been defined in detail by the design. The Figure 10 sequence diagram shows the internal details involved in passing message to a component using local transport mechanism.
The Figure 11 sequence diagram shows the internal details involved in using a component via DSS.
Figure 11. Client DSS Component Communication

DEVICE ABSTRACTIONS

The framework has device abstractions for stage controller and camera devices. The device abstractions for a device define their state, config data, state data, main messages port, internal messages port and an abstract
component class for the device. The abstract component class implements the structure of the FSM for the device and has abstract methods for implementation of the behavior. The concrete device implementations derive from the abstract component class and implement the abstract methods. The steps needed to implement device abstractions are given in Appendix A.

Figure 12 describes the state machine for the stage controller device abstraction. The stage controller abstraction mandates that any concrete implementation of stage controller must implement the position move and homing functionalities. The framework currently has concrete implementation for Zaber stage controllers.

![Stage Controller Abstraction FSM](image)

**Figure 12. Stage Controller Abstraction FSM**

Figure 13 describes the state machine for the camera device abstraction. The camera abstraction mandates that any camera implementation must implement the start acquisition and stop acquisition functionalities. The framework currently has concrete implementation for IMAQdx compatible cameras.
Figure 13. Camera Abstraction FSM
CHAPTER 4
FRAMEWORK VALIDATION AND EVALUATION

The framework was incrementally built and tested. As the first step of the framework development, the CODA component, and the FSM and networking infrastructures were developed and tested. After that the camera abstraction and IMAQdx camera implementation were developed and tested for both local and remote usage. This was followed by the development and testing of the stage controller abstraction and Zaber stage controller implementation for both local and remote usage. Finally, the Imaging Component was built and tested for both local and remote.

The first section of this chapter describes the framework validation. The second section describes the imaging component as the test case application for evaluating the framework and highlighting its advantages.

FRAMEWORK VALIDATION

The validation of the framework was done at each major step of its design and development and is explained in detail below.

Base Component and Infrastructure Test

The first step in the development process of the framework was the design and development of the base component, and the FSM, message passing and networking infrastructures. A “test component” was developed to test these out. The “test component” modeled a simple FSM with some actions and tasks, and was tested for both local and remote usage. The following Figure 14 is the screenshot of the application that was developed to test the “test component”.

53
Once the base component and infrastructure were tested, the camera abstraction was designed and developed. Some of the major functionalities exposed by the abstract camera component were to start and stop acquisition of images and getting the images captured by the camera. Once the abstract camera component was developed, the IMAQdx camera component, which is nothing but the implementation of the abstract camera component for IMAQdx compatible cameras, was designed, developed and tested. Figure 15 shows the screen shot of the application that was written to test the IMAQdx Camera component. As could be seen from the screenshot, functionalities such as searching for IMAQdx compatible cameras, starting and stopping acquisition of images, getting latest images and displaying to the user, have been tested.
As the next step, the stage controller abstraction was designed and developed. The major functionalities that are exposed by the stage controller component are position move, velocity move, relative position move and homing. This was followed by the design and development of the stage controller for the Zaber implementation of the abstract stage controller. Figure 16 shows the screenshot of the application that was developed to test the stage controllers.
Figure 16. Stage Controller Component Test

FRAMEWORK EVALUATION

The test case application that explains the advantages of the framework is the imaging component that was developed as a CODA Component and utilizes two components (IMAQdx Camera and Zaber Stage Controller) that are available in the current framework. The test case illustrates how these two components can be combined to create an imaging component that utilizes the functionality provided by both components to perform some higher level functionality; in this case, implementation of focus adjustment and auto-focusing functions. The test case also consists of a ‘Test Program’ that is a User Interface (UI) program developed in C#, used to demonstrate the use of the Imaging Component, as well as test it’s functionality. The two sub components of the test case are created on two different machines and accessed from the test program running on a third machine. The following were the benefits when developing the imaging component using the CODA framework. For a more detailed documentation
of the Imaging Component at the implementation level, please refer to the API reference guide [28].

**Benefits in Development**

The framework with its FSM and networking infrastructures, and device abstractions saved many weeks of development time. In particular, the following were the benefits that were experienced during the development of the Imaging Component.

- Since the FSM infrastructure code was pre-built and tested, it eliminates the need to develop and test flow control code for the Imaging Component thereby saving days of development time. Having a CODA FSM Base Class to derive from facilitates this process.
- Since the framework already contains Camera and Stage Controller CODA Components, it eliminates the need to develop driver and device interface code that would be required to use these devices in an imaging component; this can potentially save weeks of development time.
- Having a pre-built and tested networking infrastructure code eliminates the need to develop network communication software that would be required to use components remotely, over a network; this can potentially save weeks of development time.

**Proper State Machine Behavior**

The proper working of the state machine for the Imaging Component is extremely crucial for the correct execution of complex automation tasks. It was observed that the framework executed the state machine in a correct manner. In particular the following were observed when testing the state machine behavior for the Imaging Component.
• The test program starts up by creating an instance of the sub-components on their respective machines and then displaying their current state (shutdown)

• After program startup, the state of the UI command buttons (i.e. enabled, or disabled and ‘greyed’) is solely determined by the current state of the test components, not user actions.

• The initial state of the test components is ‘shutdown’, thus, only the “Configure’ command button is available for selection. When the “Configure’ command button is pressed, the test components are set to their default configuration. At this time this is reflected in the test program’s state display, as well as what command buttons are enabled.

• At this time, when the ‘Open’ command button is pressed, the constituent components of the imaging component perform their respective ‘Open’ operations, thus ‘opening’ the Imaging component. Upon successful opening of the components, the test program indicates that the sub-system is in the ‘Ready’ state by illuminating the ‘Opened’ LED display.

• Once the sub-system is in the ‘Ready’ state the ‘Start Acquisition’ button is enabled, at this time the user can press this button to start acquiring images from the Camera component. After successful image acquisition initiation, the display is updated with real time images from the Camera Component, and the appropriate command buttons are enabled. At this time, the imaging component’s ‘Focus Forward’, ‘Focus Back’, and ‘Auto-Focus’ may be executed, by actuating the appropriate UI command button.
Figure 17 shows the screenshot of the UI.

Figure 17. Imaging Component Test

Responsiveness

One of the main requirements of the imaging component was its ability to be responsive and in particular its ability to send notifications while it is executing another operation in parallel. The following were the observations made with respect to responsiveness of the imaging component.

- After program startup, the current state of the imaging component is sent to the test program via the CODA Component static notification mechanism. These notification results are used to update the test program UI, e.g. the state of the UI command buttons (i.e. enabled, or disabled and ‘greyed’).
- Stage controller position is displayed in real-time, based on the data sent in the ‘State Data Change’ notifications.
• Image data can be received by either dynamic notification (GetLastImage, result data) or static notification (subscribe for new image notifications).

• Results of commands, sent by actuating the UI command buttons, are received as dynamic notifications and displayed via a pop-up dialog.

**Distributed Components**

One of the major requirements in the test case was to develop the networking infrastructure needed to make the Imaging Component along with its sub-components, the camera and the stage controller to be accessible remote. This was very crucial to demonstrate the scalability and robustness of the framework. The development and testing of a very efficient networking infrastructure typically takes weeks of time. The framework prevented the need for that. In particular, the following were observed.

• The stage controller component resides on one computer, the camera component resides on a second computer, the imaging component resides on a third computer and the UI resides on a fourth computer. This illustrates how resources can be divided amongst different computers.

• Despite the inevitable network latency issues that is typically expected, acceptable image acquisition frame rates were observed (5 frames/sec), despite the fact that the images from the camera being used are quite large (on the order of 1.5 MB/image).

**Collaboration and Synchronization**

The successful collaboration and synchronization between the stage controller and camera components is extremely essential for the imaging component to perform complex automation tasks. This should work the same
irrespective of whether all of these components are local or remote with respect to one another. The following was observed with respect to collaboration and synchronization.

- After successful image acquisition initiation, the test program display is updated with real time images from the Camera Component. At this time, the imaging component’s ‘Focus Forward’, ‘Focus Back’, and ‘Auto-Focus’ functions may be executed. Note that while the selected function is executing, the test program will provide real-time updates to the stage controller position and camera image displays - demonstrating how the UI display, component notifications, and test functions, are all executing asynchronously and independent of one another.

- The focus adjustment functions (‘Focus Forward’ and ‘Focus back’) illustrate synchronization and collaboration between multiple components by performing the following operations, making use of the stage controller and camera sub-components:
  - Send command to stage controller component to move the requested amount (+ for ‘Focus Forward’, - for ‘Focus back’).
  - Wait for motion to stop, using stage controller notification.
  - Obtain the next n images that have been acquired from the camera component, over a 500 milliseconds time period.
  - Calculate the focus quality score for each image, and average them to obtain the focus adjustment result score.
  - Return the result as a notification containing the resulting focus quality score.
• The Auto Focus function illustrates a higher level of synchronization and collaboration between multiple components by performing the following operations:
  • Move stage controller to starting point.
  • Begin making focus adjustments, using the desired step size and direction until 2 successive results are less than the previous result (i.e. the last 2 focus quality scores are less than the previous one).
  • Calculate position that is the middle of the peak focus quality scores, or just determine the position that had peak focus quality score.
  • Move the stage to the calculated point.
  • This function is a very crude auto-focus implementation, and is used solely for illustrative purposes, to demonstrate the way in which CODA Components can be used in a collaborative fashion to achieve a higher level functionality.

**Extensibility**

Although the Imaging Component was tested using an IMAQdx compatible camera and a Zaber Stage Controller, it was developed in such a way that it can work with any camera that implements the camera device abstraction and any stage controller that implements the stage controller device abstraction. This is extremely powerful since it allows for development of components that are not tied to specific device drivers. Also, the imaging component was also designed in such a way that it could be derived to create new components with additional functionalities that could be accessed local as
well as remote. The framework has been designed in a manner that allows for unlimited extensibility.
CHAPTER 5
CONCLUSION

Component Oriented Distributed Architecture (CODA) has been developed. The architecture is based on the concepts of reusable components that can be distributed across multiple computers, and used in concert to develop lab automation software that is distributed, thread-safe and responsive. CODA Framework has also been developed based on the concepts of the architecture and is used for developing software that implements CODA. CODA Framework provides the requisite infrastructure that a developer can use as a starting point, to build a robust, extensible, distributed automation system.

The core of the CODA Framework is a set of .NET base classes that provide the basis for developing reusable asynchronous components that implement consistent state machine architecture. The framework ensures the state machine operates in a thread safe manner, as well as providing an interface decoupling mechanism that simplifies distributing, and using, the components across remote computers. The framework also provides a set of components that provide software abstractions for some of the more common hardware devices found in laboratory automation systems. These components allow the developer's software to be extensible, since the components greatly simplify the integration of new hardware into the software (provided an abstract component exists for the type of device the developer wishes to add).

The framework heavily leverages off of the power provided by the Microsoft Robotics Developer Studio (MSRDS): it makes use of the DSS toolkit to provide the asynchronous distributed computing capability required by the
component framework; it also makes extensive use of the CCR toolkit as the means for providing thread safe, asynchronous behavior, and inter-component message passing.

**FUTURE WORK**

The framework currently has components to control cameras that are compatible with IMAQdx and a stage Controller that is manufactured by Zaber. More components shall be added to control other camera types such as QCam and also other mechanical stages such as HSI and Maxon. Also, components to control other commonly used laboratory automation devices such as Laser shall be added. The framework makes it very easy for new components to be added. In particular, the creation of a component to control a new type of camera and a new type of stage controller is easier since there are abstractions that are currently available for the camera and stage controller components.

The framework currently uses a UML state machine composed of simple states. Although, this can model most of the state machine scenarios very effectively, there is however some advanced scenarios that could be modeled more effectively using hierarchical state machines. Hence, the current state machine design shall be enhanced to support hierarchical state machines that can be used to model advanced scenarios effectively.
REFERENCES


Lab automation and robotics: Automation on the move

[3] Software responsiveness and tips to improve it. Available from:


[5] Deadlocks and race conditions. Available from:

www.dcs.ed.ac.uk/teaching/cs3/os/slides/ipc.ppt

[7] Shared memory interprocess communication. Available from:
http://www.cs.cf.ac.uk/Dave/C/node27.html

Architectural paradigms for robotics applications
Advanced Engineering Informatics, 24 (1) (2010), pp. 4–13

<http://www.omg.org/spec/CORBA/>

[10] Microsoft, Component Object Model. Available from:
<http://www.microsoft.com/com/default.mspx>

[11] Apache, Jini also called as Apache River. Available from:
<http://river.apache.org/>

[12] Roger L. McIntosh.
Open-Source Tools for Distributed Device Control Within a Service Oriented Architecture
Journal of the Association for Laboratory Automation, Volume 9, Issue 6, December 2004, Pages 404-410


[15] OpenMP Architecture Review Board, OpenMP. Available from:
<http://openmp.org/wp/>
[16] Message Passing Interface. Available from: 
<http://www.mcs.anl.gov/research/projects/mpi/>

[17] Microsoft, Task Parallel Library. Available from: 

[18] Ericsson, Erlang Programming Language. Available from: 
<http://www.erlang.org/>

[19] Microsoft, Microsoft Robotics Developer Studio. Available from: 
<http://www.microsoft.com/robotics/>

[20] Microsoft, Concurrency and Coordination Runtime. Available from: 


[22] Microsoft, Case Studies for Microsoft Robotics Developer Studio. Available from: 
<http://www.microsoft.com/robotics/default.aspx#CaseStudies>


[25] List of hardware supported by MicroManager. Available from: 
<http://valelab.ucsf.edu/~MM/MMwiki/index.php/Device_Support>


[27] Interchangeable Virtual Instruments. Available from: 
<http://www.ni.com/ivi/>

[28] CODA Framework API Documentation. Available from: 
<https://www.dropbox.com/sh/ar90p6i6hnntbh4n/oCXrt9HD_1>

67
Appendix A

CREATING CODA COMPONENTS

This section provides the steps that need to be followed to create a new CODA component using the framework. For detailed information at a code level, please refer to the detailed API documentation [28].

All the CODA components must derive from the generic CodaComponent abstract class. The CodaComponent class takes types for state, configuration data, state data, main messages port and internal messages port and each of these items have base classes to derive from. The CodaComponent also has some abstract methods for doing the basic state machine that the derived components need to implement. Creating specialized components from already existing components such as the Imaging component or implementing device abstractions such as that for a camera component is exactly the same as creating a new CODA component except that the state, state data, config data, main operations port and internal operations to derive from is based on the component from which this component is being specialized. Also, to make the component to be able to be accessed remotely, the DSS client and DSS services needs to be derived from the appropriate DSS clients and DSS services of the corresponding base component by following the exact same steps described above. The steps for creating a new CODA component are given below.

1) Create State: The new component state must derive from CodaComponentState and must add constants that uniquely identify the different primary states of the component. As part of being a CODA component, the component already has some basic primary states which are Shutdown, ConfiguringFromFile, Configured, Opening and Ready. When assigning constants for the states, the creator of the component must ensure that
the constant value starts with one more than the ending value for the component it
derives from (in this case the value should start from five).

```csharp
public class TestComponentState : CodaComponentState
{
    public const int A = 5;

    public const int DoingTask1 = 6;

    public const int DoingTask2 = 7;
}
```

2) Create Config Data: A new class to store the configuration data for the new component has to be created. The configuration data is part of the state data. The configuration data should be marked as `[Serializable]` so that it could be cloned and passed to clients. The configuration data must implement the interface `ICodaComponentConfigData` and implement the `ExtraSerializationTypes` method. The `ExtraSerializationTypes` returns the extra types that need to be passed to the XML serializer if this class were to be used as the configuration data for the component. This information is needed to load the configuration information from the file. The configuration data for the test component looks like this.

```csharp
[Serializable]
public class TestComponentConfigData : ICodaComponentConfigData
{
    private static Type[] _extraTypes = new Type[] { typeof(TaskConfigInfo) };
    public TaskConfigInfo Task1ConfigInfo;
    public TaskConfigInfo Task2ConfigInfo;
    public virtual List<Type> ExtraSerializationTypes
```
3) Create State Data: A new class for the state data needs to be created. The state data consists of all the data for the component. The state data is composed of the state of the primary state of the component, the extended state data, the configuration data and other state data. The state data for the component must derive from the generic state data class CodaComponentStateData< TConfigData> where TConfigData must be the configuration class of the component (in the case of the test component it is TestComponentConfigData). Just like the configuration data, the state data too must be marked serializable so that it could be serialized and sent across to remote subscribers. The state data must also override the StateEnumType property and return the type of the state class which in this case is TestComponentState. This is done to link the StateData class to the State class.

```csharp
[Serializable]
public class TestComponentStateData : CodaComponentStateData<
    TestComponentConfigData >
{
    public long Task1Time;
    public bool Task1GenerateError;
    public double Task1Progress;
    public long Task2Time;
```
public bool Task2GenerateError;

public long Task2Progress;

protected override Type StateEnumType
{
    get
    {
        return typeof(TestComponentState);
    }
}

4) Create Messages: All the API messages and internal messages for the component need to be created. API messages are the ones that are posted by the outside world and internal messages are the ones that are posted by the component itself. Note that the outside world can only post those API messages to the component that are mentioned as part of the component contract. Any message that can be posted to a component should have at least three fields, and they are the message body, the acknowledgement port and the OPC port. The methodology for creation of both the API messages and the internal messages are the same except that the message body for the API messages should be marked serializable.

Create default body type and default OPC message: These kinds of messages should derive from the CodaComponentMessage class. Such a message will have a default message body which is of type CodaComponentMessageBody and a OPC port that accepts a default result. The following is an internal message having a default body type and a default OPC result.

class Task1Success : CodaComponentMessage
{
}

71
Create custom body type and default OPC message: This kind of message will have a custom body and a completion port that accepts a default response type messages. The custom body type should derive from CodaComponentMessage<TBody>. The body of the message must derive from CodaComponentMessageBody and must have a default parameterless constructor. An example of such an API message is given below.

```csharp
[Serializable]
public class SetTaskTimeRequest : CodaComponentMessageBody
{
    private long _taskTime;

    public SetTaskTimeRequest()
    {
    }

    public SetTaskTimeRequest(long taskTime)
    {
        _taskTime = taskTime;
    }

    public long TaskTime
    {
        get { return _taskTime; }
        set { _taskTime = value; }
    }
}

public class SetTask1Time : CodaComponentMessage<SetTaskTimeRequest>
{
    public SetTask1Time()
    {
    }
```
public SetTask1Time(SetTaskTimeRequest setTask1TimeBody)
    : base(setTask1TimeBody)
{
}

Create a message having custom body type and custom OPC: This kind of message will have a custom body and a OPC port that accepts a custom OPC message. An example of such a message is given below.

class CustomBody : CodaComponentMessageBody
{
    public int SomeData { get; set; }
}

class CustomMessage : CodaComponentMessage<CustomBody, FunctionResult<int>>
{
}

Create Get Data messages: The Get messages of the component must derive from the abstract GetData generic class as given below.

    public sealed class GetTaskTime : GetData<long>
    {
    }

Create Subscribe messages: The subscribe messages of the component must derive from the abstract Subscribe generic class as given below.

    public sealed class SubscribeToTask1Progress : Subscribe<double>
    {
    }

5) Create Main Messages Port

The main messages port consists of the API messages that could be posted to the component. The main messages port must derive from the generic
CodaComponentMainMessagesPort<TConfigData, TStateData> class specifying the correct state data and config data types for the component. The main operations port must have a public parameterless constructor that calls the base class constructor with the API messages for this component. If the main messages port is part of a component that can have derived components, then it should expose a parameterized constructor that takes the API messages for the derived component and should append it with the API messages for this component and pass it on to the base class constructor parameterized constructor. The main messages port for the test component is given below.

```csharp
class TestComponentMainOperationsPort : CodaComponentMainMessagesPort<TTestComponentStateData, TTestComponentConfigData>
{
    private static Type[] _messageTypes;

    static TestComponentMainOperationsPort()
    {
        _messageTypes = new Type[] {
            typeof(GotoA),
            typeof(ReturnToReadyState),
            typeof(DoTask1),
            typeof(DoTask2),
            typeof(SetTask1Time),
            typeof(SetTask2Time),
            typeof(Task1GenerateError),
            typeof(Task2GenerateError)
        };
    }
```
6) Create Internal Messages Port: Internal messages port contains those messages that are posted from within the component. The internal messages port should derive from CodaComponentInternalMessagesPort and should follow the same guidelines as the main messages port. The internal messages port for the test component is given below.

```csharp
public class TestComponentInternalMessagesPort : CodaComponentInternalMessagesPort
{
    static TestComponentInternalMessagesPort()
    {
        _messageTypes = new Type[] {
            typeof(Task1Success),
            typeof(Task2Success),
            typeof(Task1ProgressStatusUpdate),
            typeof(Task2ProgressStatusUpdate),
            typeof(Task3StatusReport),
            typeof(Task3Success),
            typeof(Task4StatusReport),
            typeof(Task4Success),
            typeof(Task5StatusReport),
            typeof(Task5Success)
        };
    }
}
```
typeof(Task2ProgressStatusUpdate)

};

}  

public TestComponentInternalMessagesPort()
    : base(_messageTypes)
{
}

protected TestComponentInternalMessagesPort(Type[] newMessageTypes)
    : base(newMessageTypes)
{
}

}  

7) Create Component: A new component must derive from the generic CodaComponent<TConfigData, TStateData> class by supplying the types for the state data and config data. The new component must have a public constructor that takes in an object of CodaComponentCreationInfo and call the protected constructor of the base component and pass in the creation information. There are scenarios where it may be desired that the component to be extended. In this scenario a protected constructor must be created that takes in the main messages port and internal messages port and pass them on to the base component protected constructor.

public class TestComponent : CodaComponent<TestComponentStateData, TestComponentConfigData>
{
    public TestComponent(CodaComponentCreationInfo creationInfo):
A) Building the state machine: The state machine of a component has a list of states, triggers and transitions. The trigger for a state machine is nothing but a message that is posted to the component. Note that all triggers are messages but not all messages are triggers.

Creating primary states: The primary state can be a static primary state or doing primary state. A static primary state is created by making use of the non-generic PrimaryState class. An example of a simple static primary state is given below. The primary state must be mapped to the corresponding state constant defined
previously (in this example TestComponentState.A). Other constructors can be used to specify the state entry and state exit handlers.

```csharp
var stateA = new PrimaryState(TestComponentState.A);
```

A doing primary state is like a static primary state except that it has to have a state handler that gets executed asynchronously. An example of a doing primary state is given below.

```csharp
var task1DoingState = new PrimaryState<DoTask1>(
    TestComponentState.DoingTask1, DoingTask1StateBehavior);
```

In this example DoingTask1StateBehavior is the state behavior for the doing state and it is an iterator that takes DoTask1 message as the argument. The state behavior is executed asynchronously and is responsible for sending the OPC message to the completion port of the message indicating whether the operation resulted in success or failure and pass on any data if needed. Note that the state behavior cannot update the state data directly and can only do so by posting a message and causing a transition. The state behavior looks like the following

```csharp
private IEnumerator<ITask> DoingTask1StateBehavior(DoTask1 doTask1)
{
}
```

Create Transitions: Transition is something that is fired by the state machine in response to a trigger message. Multiple transitions for a state can be added. Transitions are classified as doing either of two kinds of operations, which are actions and starting tasks. If the operation done by the transition is an action, then the OPC is sent by the state machine engine. However, if the operation done by the transition is starting a task, the user of is responsible for sending the OPC. Transitions can
optionally have guard condition, extended state data handler and the property to ignore errors if they arise in its execution. All transitions must have a target state except for internal transitions which do not have a target state.

Example of a transition is given below. The transition defined below means that when "GotoA" message arrives in ReadyState, the transition by the Transition by the name "GotoA" will be fired and will cause the component to go to state “A”.

```csharp
```

Example of a transition to a doing state

```csharp
stateA.AddTransition(new Transition<DoTask1>("DoTask1", OperationType.StartTask, task1DoingState));
```

Internal transitions are special kinds of transitions that only have a transition behavior and does not involve a target state. An example of an internal transition is given below.

```csharp
task2DoingState.AddTransition(new Transition<TaskProgressUpdateBody, Task2ProgressStatusUpdate>("Task2ProgressStatusUpdate", OperationType.Action, Task2ProgressUpdateHandler));
```

Transition behaviors are normal message handlers as given below.

```csharp
private void Task2ProgressUpdateHandler(Task2ProgressStatusUpdate task2ProgressStatusUpdate)
{
}
```
If there is any error in the transition behavior and the transition is to be abandoned, an exception needs to be thrown from within the transition behavior. In this scenario, the state machine engine will abandon the transition.

```csharp
private void Task2ProgressUpdateHandler(Task2ProgressStatusUpdate task2ProgressStatusUpdate)
{
}
```

B) Add Argument Validation: The arguments of the message can be validated before they are being processed by the message handlers. Make use of the `AddArgumentValidationHandler<TBody, TResponse, TMessage>` method to add argument validation. The following is the argument validation added in the base component for configuring from file message.

```csharp
AddArgumentValidationHandler<ConfigureFromFileRequest, ConfigureFromFile>(
(configureFromFileRequest) =>
{
    if (String.IsNullOrEmpty(configureFromFileRequest.FilePath))
    {
        throw new ArgumentNullException("The path of the configuration file cannot null or empty");
    }
});
```

C) Adding Property sets: Property sets involve setting of a certain piece of settable state data of the component. Property sets can be added using the genetic method
protected void AddPropertySet<TBody, TMessage>(PropertySet<TBody> propertySet)

A property set has an optional guard condition to check whether the property set is valid for the state and a property set handler that contains the actual property set implementation. An example of adding a property is given below

AddPropertySet<SetTaskTimeRequest, SetTask1Time>(new PropertySet<SetTaskTimeRequest>(
    () =>
    {
        return StateData.State != TestComponentState.DoingTask1;
    },
    (setTask1Time) =>
    {
        StateData.Task1Time = setTask1Time.TaskTime;
    })
);

D) Creating receivers for messages: All the messages whether it is an API message or an internal message must have receivers with handlers associated to them. The receivers can be either of the following

Trigger Message Handler: Trigger receivers are those that cause a change to the state of the component and its processing goes through the state machine engine. Once the state machine engine is done processing the message, the trigger receiver will send state change notifications to all state change subscribers. The code to create a trigger receiver is given below.
Arbiter.ReceiveWithIteratorFromPortSet<GotoA>(true, MainMessagesPort,
    TriggerMessageHandler<GotoA>)

Error Trigger Message Handler: Error trigger receivers are exactly like trigger receivers except that they notify a trigger which is also an error. Apart from sending state changes to the state change subscribers, an error trigger receiver also sends the subscribers for errors that an error has occurred in the component. The code to create a trigger receiver is given below. Note that error trigger receivers must be used only for error trigger messages and for normal trigger messages, make use of trigger receivers

Arbiter.ReceiveWithIteratorFromPortSet<Error>(true, InternalMessagesPort,
    ErrorTriggerMessageHandler<ErrorBody, Error>)

PropertySet Message Handler: Property set receivers are used to handle property set messages. The messages do not have to go via the state machine processing. An example for creating a property set receiver is given below. Property sets also involve state change and state change notifications are sent once the property set is done.

Arbiter.ReceiveFromPortSet<SetTask1Time>(true, MainMessagesPort,
    PropertySetMessageHandler<SetTaskTimeRequest, SetTask1Time>)

GetData Handler: The user needs to write their own handlers for all the GetData messages. Inside those handlers they can make use of the GetDataHandler method in the base component to help send the ACK and OPC to the caller.

Subscribe Handlers: Just like GetData handlers, the user needs to write their own handlers for all the Subscribe messages. Inside those handlers they can make use of the SubscribeHandler method in the base component to help send the ACK and OPC
to the caller and also to add the notification port to the list of notification ports for this subscription.

Other Handlers: The handlers for any message will mostly fall under any of the above categories. However, if there is a message whose handler does not fall in the above three categories, the user needs to write his own handler.

Important Note: It should be ensured that when creating the receivers the right Portset should be given. When creating a receiver for an internal message pass the Portset as InternalMessagesPort and when the receiver for main operations port is created, the Portset should be given as MainOperationsPort. Failure to do that will cause a run time exception which will be extremely hard to debug.

E) Adding receivers to the all tasks operations Interleave: The receivers for all these messages should be added as part of the exclusive group of the messages interleave for the component. An example is given below

```csharp
AllTasksInterLeave.CombineWith(new Interleave

    (
        new ExclusiveReceiverGroup

        (  
            Arbiter.ReceiveWithIteratorFromPortSet<GotoA>(true, MainMessagesPort, TriggerMessageHandler<GotoA>),
            Arbiter.ReceiveWithIteratorFromPortSet<Task1Success>(true, _internalMessagesPort, TriggerMessageHandler<Task1Success>)
        ),
        // Other receivers
        new ConcurrentReceiverGroup()
    )

83```
F) Implementing base component state behaviors and transition behaviors: The following methods need to be implemented in order for the new component to work.

```csharp
protected override void SetDefaultConfig()
{
}

protected override IEnumerator<ITask> OpenHandler(Open open)
{
}

protected override void Close()
{
}

protected override void Configure()
{
}
```

Note that if the new component is itself an abstract component, it can leave some of these methods empty for the derived components to implement.

G) Ignoring message types in Errored State

If the component is in errored state, we can ignore processing of a message. Once the argument validation is done, the framework checks if the component is in the errored state the message should be ignored. An example to add a message to the list of ignored messages,

```csharp
IgnoredMessageTypesInErroredState.Add(typeof(DoTask1));
```
8) Create Component Interface object: Any user of the component communicates with the component by making use of the component interface object. The component interface object has functions for all the messages that could be sent to the component and is responsible for creating the messages and posting them to the transport which is in turn responsible for forwarding to the component. The component interface object waits for the ACK from the transport, and if the ACK is an error ACK, it throws an exception to the caller. If the ACK is not an error ACK, the interface object returns the ACK result and the OPC port to the caller. To create a new component interface object, the user has to derive from the generic component interface object class and add methods for the messages of the component.

Depending on the message created for the function, the following ways to post it to the transport mechanism needs to be used.

a) If the message is a normal API message, one of the generic PostMessage methods of the base class needs to be used.

b) If the message is an API subscription message, the generic PostSubscribeMessage method of the base class needs to be used.

c) If the message is an API get data message, the PostGetData method needs to be used.

```csharp
public sealed class TestComponentInterfaceObject :
    CodaComponentInterfaceObject<TestComponentState, TestComponentConfigData, TestComponentStateData>
{
}
```

```csharp
public TestComponentInterfaceObject(ITransport Transport)
    : base(Transport)
```
public Port<CodaComponentMessageResult> GotoA()
{
    return PostMessage(new GotoA());
}

9) Create Transport Mechanism: The framework supports two transport mechanisms. One is local and another is DSS. If the component has to be accessed only local then no extra code needs to be added. However, if the component needs to be accessed via DSS as well, then the following has to be done.

A) Create DSS service base: The steps to create a new DSS service base is as follows.
   a) The DSS service base of the newly created service must derive from the generic CodaComponentDsspServiceBase class.

   class TestComponentDssService :
   CodaComponentDsspServiceBase<TestComponentState, TestComponentConfigData, 
   TestComponentStateData>
   {
   
   }

   b) The state of the newly created service must be CodaComponentDssServiceState [ServiceState]

   CodaComponentDssServiceState _state = new CodaComponentDssServiceState();
c) The operations port for the newly created service must be of the type 

\texttt{CodaComponentDssServiceOperations}. The operations ports member must be declared similar to the following.

\begin{verbatim}
[ServicePort("/TestComponentDssService", AllowMultipleInstances = true)]
CodaComponentDssServiceOperations _mainPort = new CodaComponentDssServiceOperations();
\end{verbatim}

d) The CreateComponent method needs to be implemented. In this method the component needs to be created

\begin{verbatim}
protected override void CreateComponent(string componentName, int

    numberOfDispatcherThreads)
{
    Component = new TestComponent(componentName,

        numberOfDispatcherThreads);
}
\end{verbatim}

e) The CreateMainMessageTypes methods need to be implemented. The CreateComponent method must contain the code for creating a new instance of the component. The CreateMainMessageTypes must create the message types for all the API messages, API subscribe messages and API get messages by making use of the correct overloaded CreateApiMainMessageType, CreateGetMainMessageType and CreateSubscriptionMainMessageType respectively. It should be made sure that the base class CreateMainMessageTypes() is also called, so that the main messages of the base components are also registered with the DSS.

\begin{verbatim}
protected override void CreateMainMessageTypes()
{

87
\end{verbatim}
Create the main message types of the base DSS service.

```csharp
base.CreateMainMessageTypes();
CreateApiMainMessageType<
    GotoA>();
CreateApiMainMessageType<
    ReturnToReadyState>();
CreateApiMainMessageType<
    DoTask1>();
CreateApiMainMessageType<
    DoTask2>();
CreateApiMainMessageType<
    SetTaskTimeRequest, SetTask1Time>();
CreateApiMainMessageType<
    SetTaskTimeRequest, SetTask2Time>();
CreateApiMainMessageType<
    TaskGenerateErrorBody,
    Task1GenerateError>();
CreateApiMainMessageType<
    TaskGenerateErrorBody,
    Task2GenerateError>();
```

B) Create DSS Subscription Messages

The service should also create one DSS subscription request and one notification message for every subscription message of the component. The DSS subscription request message must derive from SubscriptionNotificationRequest and the DSS notification message must derive from dssp.Update message and pass the newly created request message as the body. Note that both the request and notification messages must have a [DataContract] on it.

```csharp
[DataContract]
public sealed class SampleSubscribeRequest : SubscriptionNotificationRequest
{
}
```
[DataContract]

```csharp
public sealed class SampleNotification : Update<
    SampleSubscribeRequest,
    PortSet<
        DefaultUpdateResponseType, Fault>
>
{
}
```

C) Create DSS client

A DSS client needs to be created only when there are new subscription messages introduced by the component. To create a new DSS client, derive from the generic CodaComponentDssClient and for every new subscription message, call the generic CreateDssServiceSubscription function to add the new subscription type.

```csharp
public class CameraComponentDssClient : CodaComponentDssClient<
    TestComponentState, TestComponentConfigData, TestComponentStateData>
{
    public CameraComponentDssClient()
    {
        CreateDssServiceSubscription<
            TestComponentStateData,
            TestSubscribe, SampleSubscribeRequest,
            SampleNotification>();
    }
}
```

10) Using CODA Components: The clients communicate with the component by making use of the interface object for the component. In order to use the component, the component needs to be created first. Then, the transport is created and bound with the component. Finally, the interface object needs to be created and bound with the
transport. The interface object is then used by the clients to communicate with the component.

For local components, the following is done

```csharp
TestComponent testComponent = new TestComponent(new CodaComponentCreationInfo("TestComponent");
var localTransport = new LocalTransport(testComponent);
TestComponentInterfaceObject = new TestComponentInterfaceObject(localTransport);
```

For remote components, the following is done.

```csharp
var dssClient = new CodaComponentDssClient<
  TestConfig1,
  TestComponentStateData>();
dssClient.createComponent(new CodaComponentCreationInfo("TestComponent", new IPEndPoint(IPAddress.Parse("127.0.0.1"), 40001)),
  "http://schemas.tempuri.org/2012/09/testcomponentdssservice.html");
var testComponentInterfaceObject = new TestComponentInterfaceObject(dssClient);
```

Once the component has been used, all the resources need to be disposed. This is done by calling the Dispose method in the interface object. The dispose method in the interface object will make sure that apart from disposing itself, it disposes the transport and the component as well.