Discovery Channel Telescope software key technologies

Paul J. Lotz*
Lowell Observatory, 1400 W. Mars Hill Road, Flagstaff, AZ 86001

ABSTRACT

The Discovery Channel Telescope (DCT) is a 4.3-meter astronomical research telescope being built in northern Arizona as a partnership between Discovery Communications and Lowell Observatory. The project software team has designed and partially implemented a component-based system. We describe here the key features of that design (state-based components that respond to signals) and detail specific implementation technologies we expect to be of most interest: examples of the Command Pattern, State Pattern, and XML-based configuration file handling using LabVIEW classes and shared variables with logging and alarming features.

Keywords: DCT, Lowell Observatory, LabVIEW class, shared variable, Command Pattern, State Pattern, publish-subscribe, telescope software, component, XML, cRIO

1. INTRODUCTION

After considering aspects of the DCT sofware's component design, including the implementation of its messaging system, we provide examples of Object-Oriented implementations, discuss the selected development environments, and summarize our experience with software engineering technologies.

2. COMPONENT DESIGN

The key system design principle we have adopted is to allocate software functionality to software components. Each component is a separate, stand-alone application. The component controller subscribes asynchronously to relevant data messages, and responds in a manner determined by the data content in combination with the controller's current state. This idea is very similar to the concept of Concurrent Components Communicating Anonymously (C^3A) the Large Synoptic Survey Telescope (LSST) team has previously described.[1]

2.1 System analysis

The team identified a hierarchy of components, each of which has a definable purpose and a distinct data boundary. The team then refined the component description by developing detailed functional and nonfunctional requirements based on user needs. In turn these led to the definition of data messages the component publishes and to which it subscribes.

We have organized the software components into two major areas: instruments and telescope systems (see Figure 1).

* paul.lotz@lowell.edu; phone: 928-233-3204; fax: 928-233-3268; lowell.edu

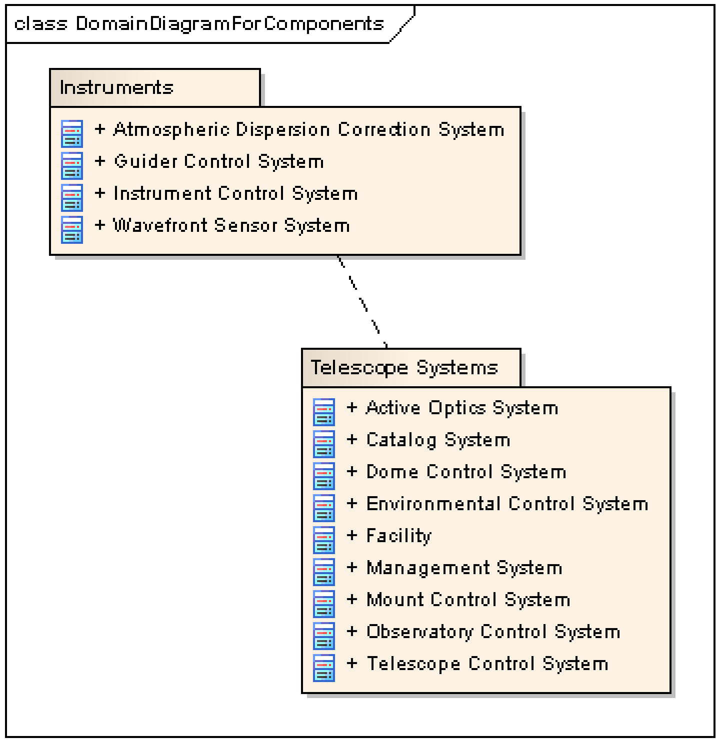

Figure 1. DCT software high-level components.

Within the telescope system, for example, the Observatory Control System summarizes the health of the various components and sends notifications of high-level state changes (see Figure 2).

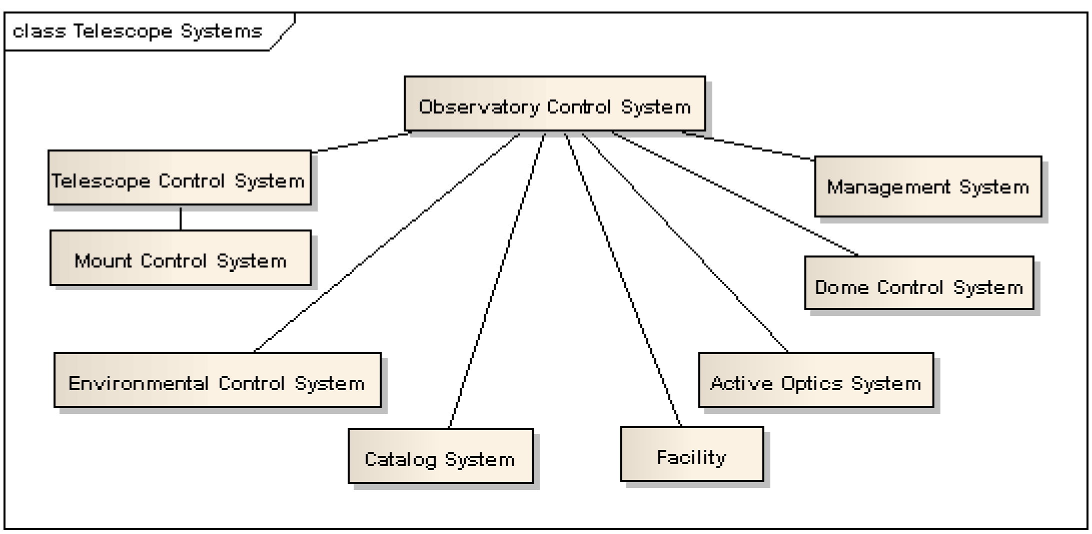

Figure 2. Components of telescope system.

2.2 Statemachine

Each component operates on its own and responds to external stimuli (data messages) depending on the current component state.

Components differ considerably in complexity and therefore in the detailed states required. Hence we define the detailed state for each component separately. The component's detailed state permits precise definition of a controller's state behavior. See Figure 3 for a list of the possible states of the primary mirror position controller.

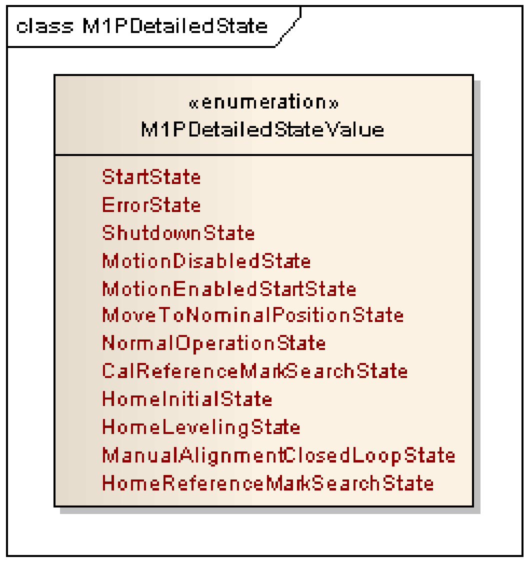

Figure 3. Primary mirror axial support position control system (M1P) states.

Information on the detailed state facilitates design and implementation, as well as debugging at the component level. Each component publishes its current state and logs this information for analysis if desired. The component's user interface and usually the component's parent component subscribe to the detailed state information.

In addition, the team has defined a set of states common to all components: Off, Standby, Enabled, Disabled, Fault. This constitutes a summary state. Figure 4 shows the summary states and the relationships between these.

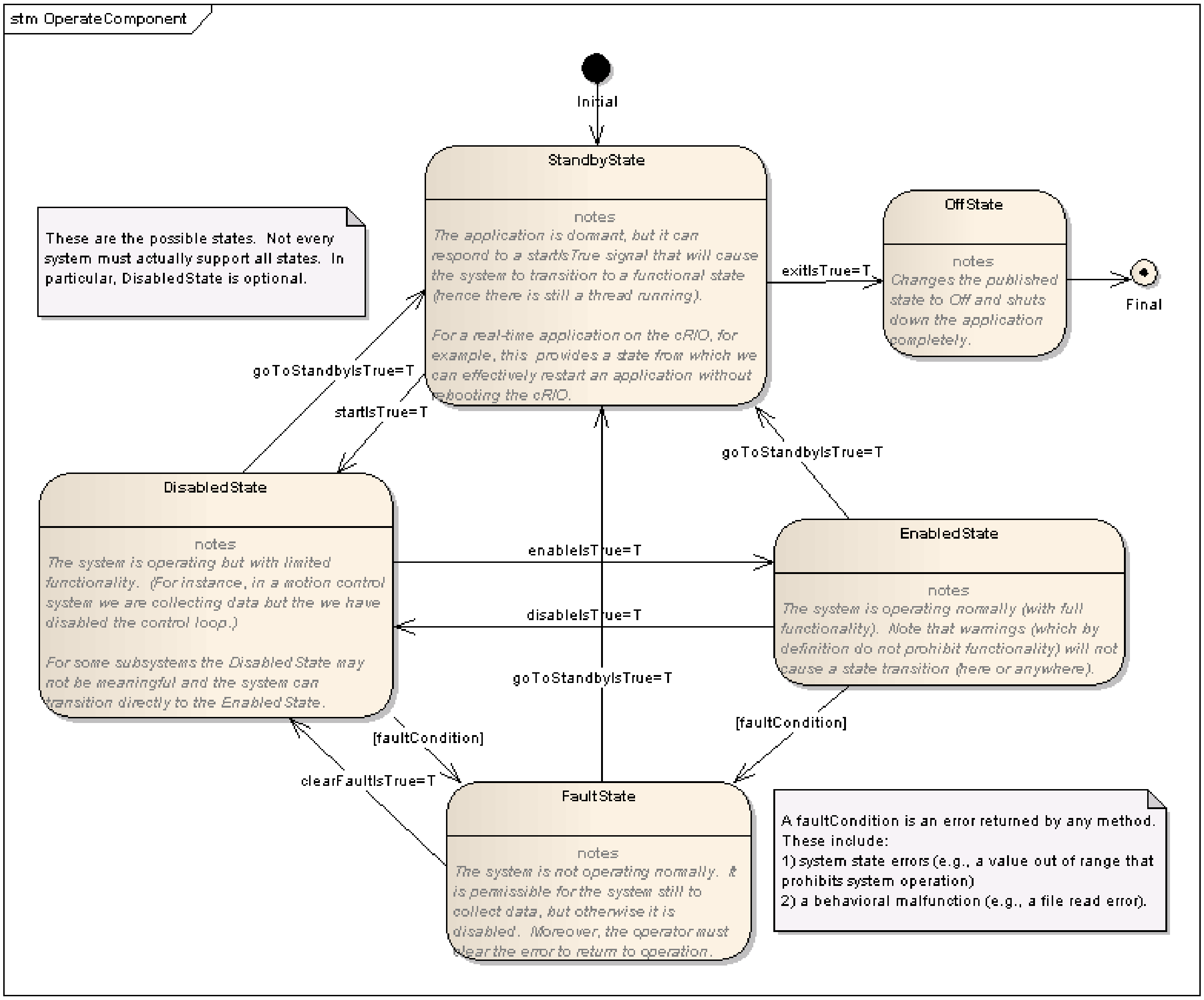

Figure 4. Summary states.

Each detailed state maps to one of these summary states. The system reports summary state at each level. The current summary state of a component depends on the summary state of each of its contained components in a hierarchical fashion.

2.3 Signaling

Each component subscribes to signals bearing data it requires for operation, and publishes data for use by other components. The team has implemented a networked publish-subscribe messaging system that represents data natively or via a common representation (XML). The message content may be simple or complex types (including objects). Being able to send objects in LabVIEW enables the use of design patterns, most notably the Command Pattern.

2.3.1 Ethernet-based technologies

The use of Ethernet-based technologies in the system allows for multi-point communication between widely spaced components. Moreover, technologies based on TCP can provide high data rates with guaranteed message delivery.

2.3.1.1 National Instrument compactRIO

In addition to networked PCs (desktop and rack-mounted computers), the system utilizes networked embedded real-time controllers in National Instruments compactRIO (Reconfigurable Input Output) modules. Each 9074 compactRIO device in the system has a 400 MHz real-time processor, as well as a 2 million gate field-programmable gate array (FPGA) that we program in LabVIEW to handle the input/output connections to the various hardware modules. The C-series hardware modules are plug-and-play.

Figure 5 gives an example of graphical coding for an FPGA target. LabVIEW uses Xilinx tools to compile the code into a bit-file deployable on the FPGA target.

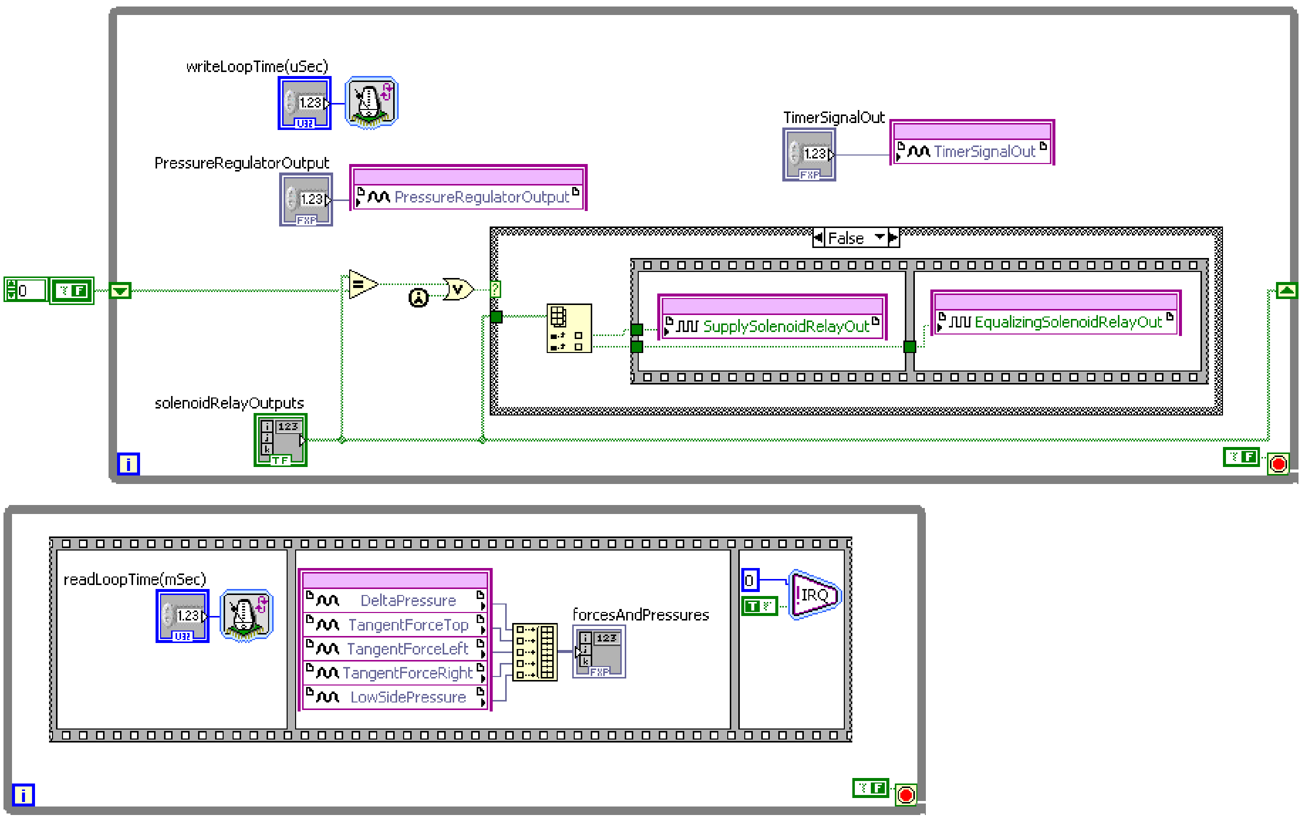

Figure 5. Primary mirror lateral supports FPGA code.

Each compactRIO communicates with the rest of the system via the same Ethernet-based communication (National Instruments networked shared variables) used for communication between all of DCT's LabVIEW-implemented components.

2.3.2 Publish-subscribe protocol

The system utilizes two implementations of publish-subscribe protocols. Each client publishes specified data messages to the outside world and subscribes to selected messages from the outside world. Messaging is asynchronous. A server manages the connections between the subscribers and publishers, so that these are decoupled from one another. Moreover, the messaging systems implemented in this project support event-based communication, eliminating the need for polling by systems that support event-based programming. (Currently the real-time controllers the team has developed in the system respond to hardware-generated interrupts and can read or write messages each loop.)

2.3.2.1 ActiveMQ

For communication between LabVIEW and nonLabVIEW components or between nonLabVIEW components DCT uses an open-source implementation of the Java Message Service (JMS) Application Programming Interface (API) called ActiveMQ produced by the Apache Foundation. JMS is a popular API and its specifications meet the system functional and performance requirements. ActiveMQ offers clients in several popular development environments. The C++ client (CMS) serves as a bridge between the server and LabVIEW.

The team has developed a customized implementation of the CMS client. The LabVIEW application accesses the CMS client (compiled as a DLL) interface via LabVIEW's Call Library Function node. LabVIEW calls methods on the CMS client to create or subscribe to message topics and to write messages to the ActiveMQ message broker. When a message arrives to which the CMS client subscribes, the client generates a LabVIEW user event that contains a string with the message content. This allows the LabVIEW application to respond to message events.

The team also tried the .NET client (NMS). At the time NMS was not a released product. The team found the interface to LabVIEW far simpler to create, but as a consequence of the managed/unmanaged interface performance was inadequate for the purpose.

2.3.2.2 National Instruments Shared Variable Engine

For communication between LabVIEW components DCT uses the National Instruments LabVIEW networked shared variable. Both desktop and real-time DCT applications read and write shared variables through a common API. DCT applications deploy shared variables on the Shared Variable Engine (SVE) server on a Windows computer.

It is possible to host the server on another platform, but the SVE only supports certain functionalities on a Windows computer. In particular, National Instruments currently supports the Datalogging and Supervisory Control (DSC) module only on Windows. The DSC module provides features such as datalogging, alarming, scaling, user-based security, shared variable event handling, and initial value specification.

An application reads or writes a shared variable using one of several available APIs. These include shared variable nodes, front panel control binding, methods on the DataSocket palette, and methods on the (new in LabVIEW 2009) Shared Variable palette.

Currently, shared variables implement a TCP communication layer. The team noted a great improvement in reliability and performance when NI transitioned from UDP-based communications.

Shared variables are configurable through a dialog particular to each shared variable or through the spreadsheet-like Multiple Variable Editor. Figure 6 shows part of this interface.

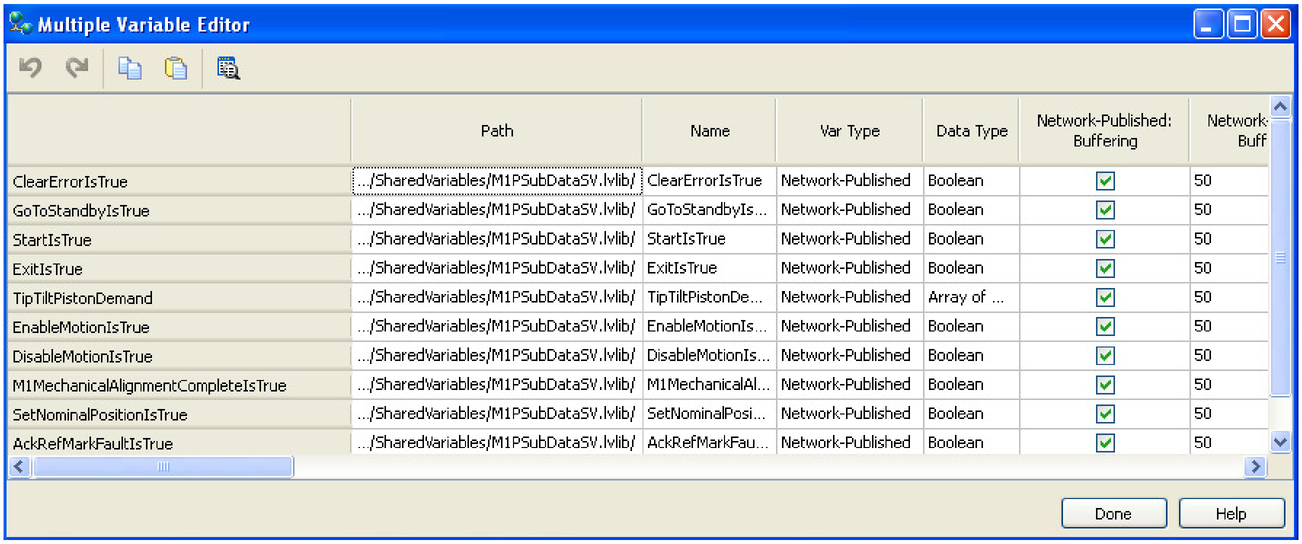

Figure 6. Editing the M1P variables in the Multiple Variable Editor.

It is possible to apply edits to shared variables dynamically.

Shared variables offer a native LabVIEW interface for sending data. We find the shared variable node interface clean and readable. Figure 7 [indicates offers an example of writing data to shared variables.

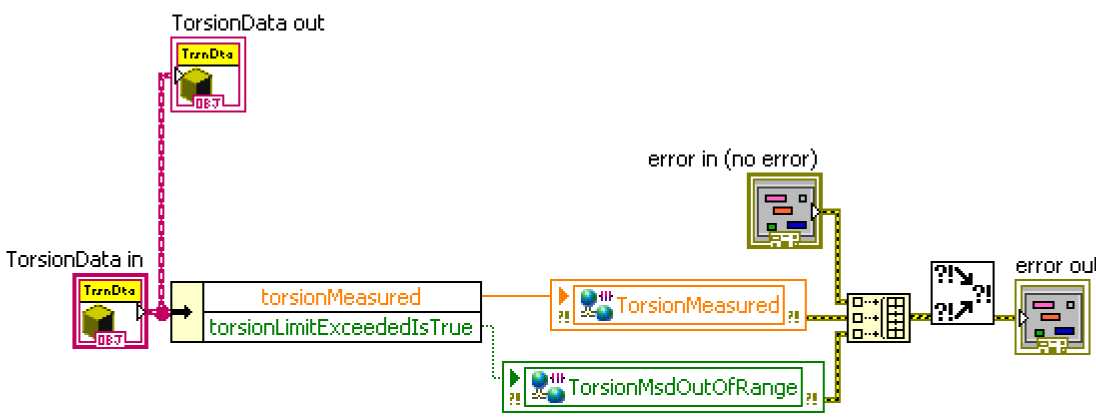

Figure 7. Writing torsion data to shared variables.

2.3.2.2.1 Shared variable events

The DSC module includes an API to register for shared variable events for use with the event structure in Windows applications. We consider this capability quite helpful to ensure the proper processing of all messages, and of course the use of events instead of polling can reduce processor usage.

2.3.2.2.2 Datalogging

The DSC module includes functionality to write shared variable values, alarms, and events to a Citadel historical database. NI provides an interface to the data through the Historical Data Viewer or programmatically with functions on the Historical palette. It is also possible to create a custom tool to access Citadel data using SQL.

2.3.3 Message content

Messages may be simple or complex in content, and may have native or universal representations.

2.3.3.1 Basic types

DCT applications send data as basic types in the following instances:

2.3.3.1.1 RT FIFO-enabled shared variables

DCT real-time applications use RT FIFO-enabled shared variables to ensure the real-time performance of the application (not of the shared variable communication, of course). RT FIFO-enabled shared variables must have a fixed size in order to support this performance, thus limiting the available types for variable definition to basic types (including arrays and Enumerations but not Strings).

2.3.3.1.2 Data used for alarming

For some applications we have configured the value-based alarming of a variable. Value-based alarms are only configurable for simple numeric and Boolean types (not including collections).

2.3.3.2 Objects

The DCT team independently invented a way to send messages as LabVIEW objects. Generally, it is possible to configure a shared variable to use a custom type, but the available custom types do not include LabVIEW objects. To circumvent this limitation, in the DCT solution the message sender flattens the LabVIEW object to a string and writes the flattened string to a String-typed shared variable. For example, Figure 8 shows part of the ConfigurationCommand class hierarchy.

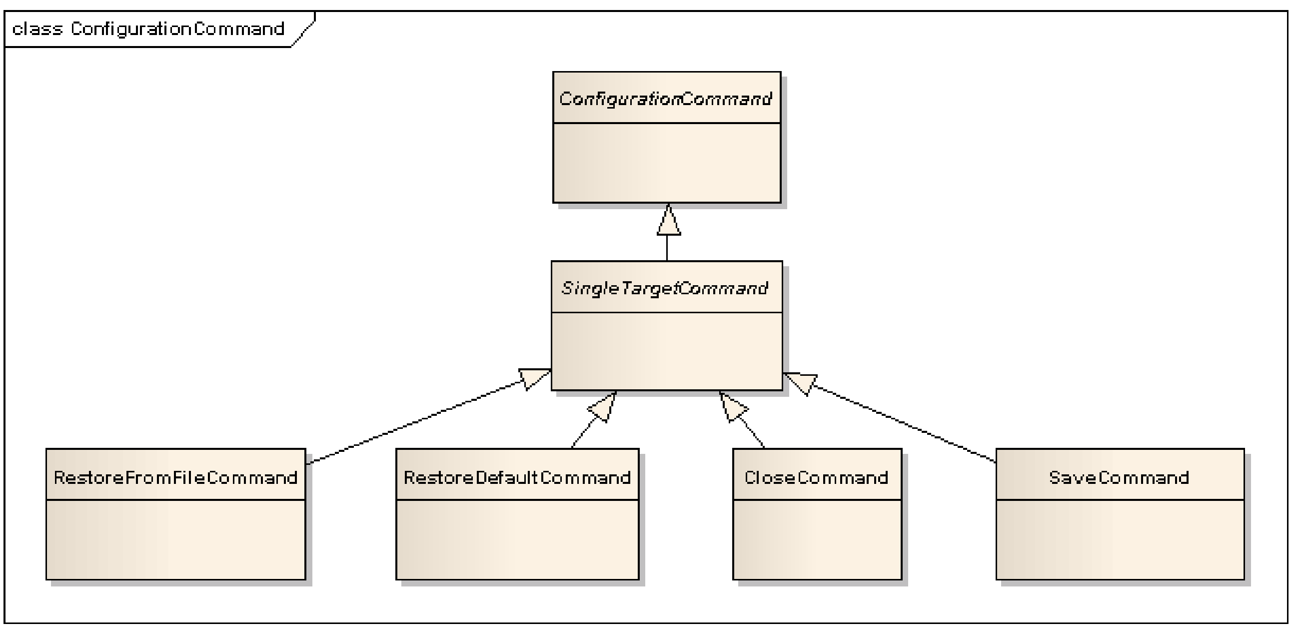

Figure 8. Simplified configuration command hierarchy.

The sender flattens the specific command (e.g., SaveCommand)—in this case to an XML string—and publishes it (see Figure 9).

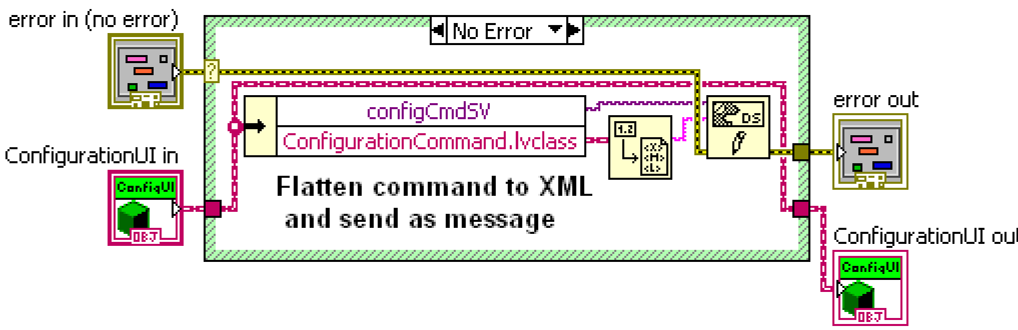

Figure 9. Publishing a flattened LabVIEW object.

Upon receipt of the message, the receiver unflattens the String-typed message back to the object. The receiver must know the parent type (not the specific type!) of the received object. Further, both the sender and receiver must have access to the class definition (see Figure 10).

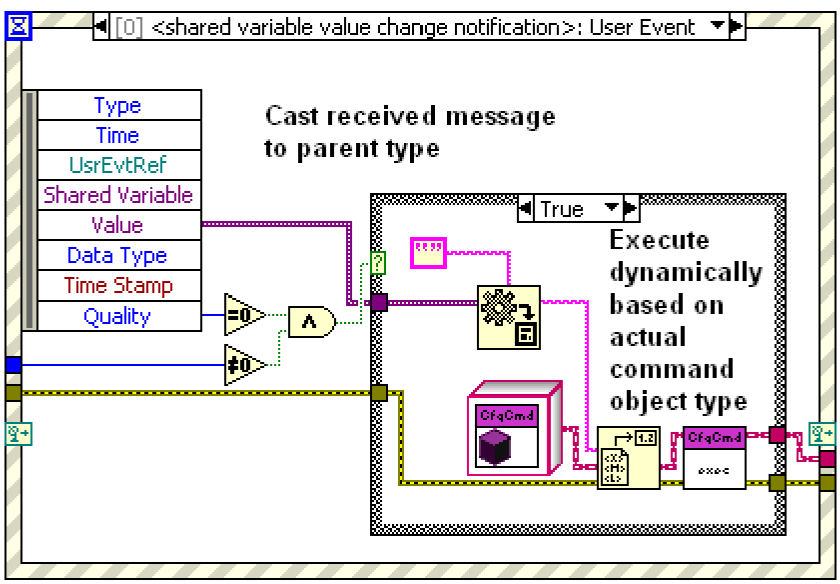

Figure 10. Casting a received message to a LabVIEW class.

In this implementation, each shared variable maps to exactly one type. Both the sender and the receiver projects incorporate the required class definitions (at least in the dependencies), which exist in the version control repository in a single location in order to support code maintainability.

2.3.3.2.1 Command pattern

As a specific instance we dig deeper into examples of the use of the Command Pattern[2] in LabVIEW.

The implementation of the Command Pattern requires that we create an abstract Command class and then descendant concrete command classes that inherit from this. (In practice intermediate descendant classes—e.g., SingleTargetCommand in the example above—are abstract as well; only the leaf classes represent invokable commands.) We have already shown one branch of the ConfigurationCommand hierarchy above. Another branch allows us to create macro commands easily by creating arrays of individual command objects (see Figure11).

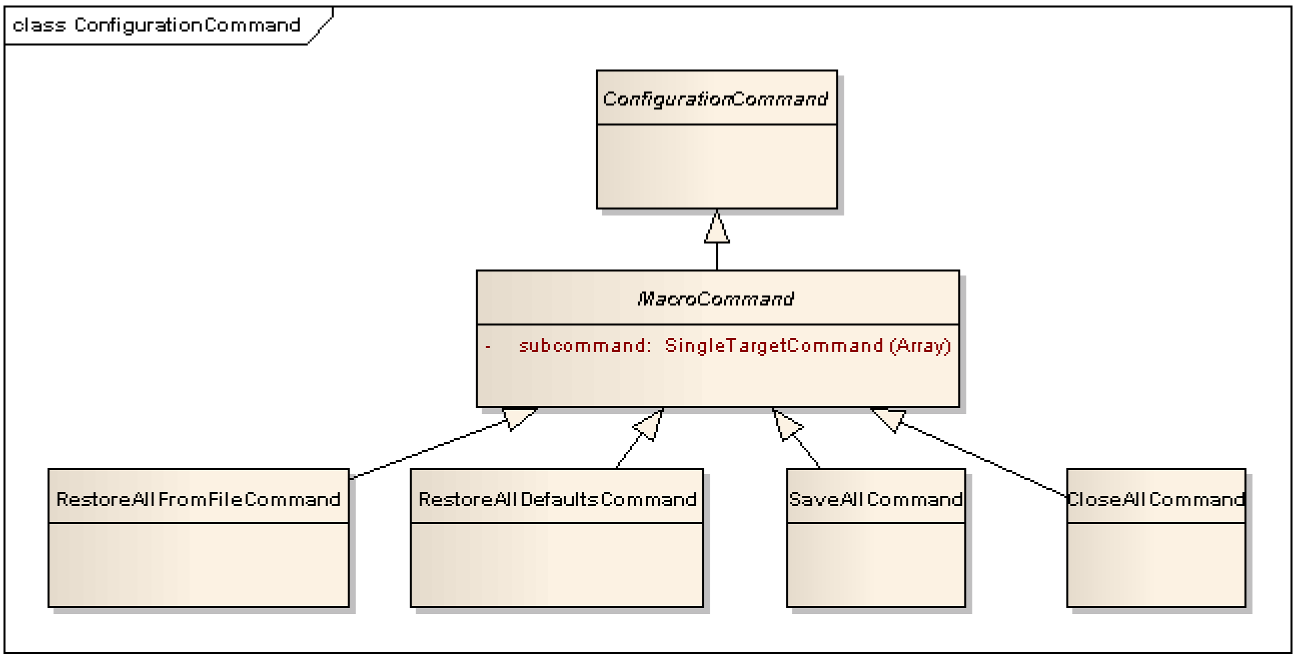

Figure 11. Macro command branch.

Further, we can define each command class with its own definition, which allows us to create a command interpreter quite easily. For instance, in another application we have defined two types of FixedTarget. The two concrete types (RADecTarget and AzElTarget) have some parameters in common (we only have to define these once on FixedTarget) and each has unique parameters (defined on the particular class). Figure 12 illustrates the FixedTarget hierarchy.

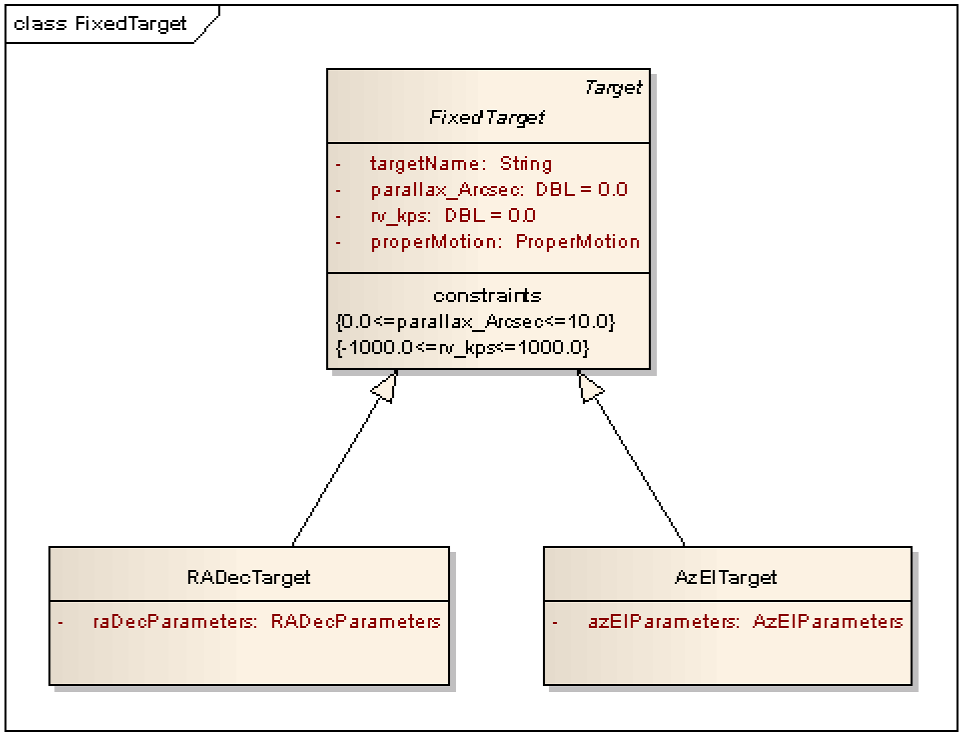

Figure 12. FixedTarget hierarchy.

We have already shown above how the receiver publishes the command and how the receiver subscribes to the message. After the receiver unflattens the message to an object it invokes a ConfigurationCommand:execute method. The actual method that executes depends on the actual message type. This is an example of dynamic dispatching (see Figure 13).

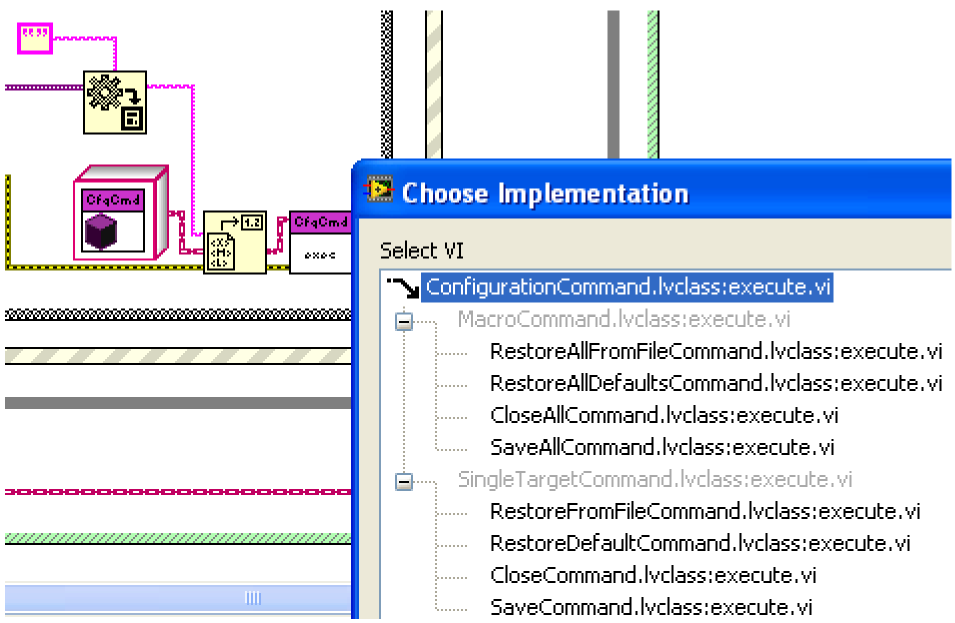

Figure 13. Dynamic dispatch of ConfigurationCommand:execute.

2.3.4 Message format

At the lower level, all TCP messages are strings, but at the application level, messages can have several formats. Within LabVIEW it is convenient to send messages retaining native type formats where possible. Where this is not possible (LabVIEW objects) flattened strings (binary strings that have meaning only within LabVIEW) are convenient. For sharing data between applications written in different development environments DCT has adopted an approach using XML encoding of information.

Endianness issues present issues with many other encoding schemes. Of course other string-encoded formats are available, such as a string command with parameters—which the DCT system does use for communication between the TCS and the MCS. XML, however, offers us the ability to encode any data (not just commands) in a structured way. This data may include objects.

2.3.4.1 XML

The DCT team designed and implemented an XML schema for representing any type of data. In its essentials the schema follows that of SimpleXML.

LabVIEW has its own XML schema (currently defined in LVXMLSchema.xsd), but this schema is quite customized and the encoding is generally only meaningful within LabVIEW. In particular, if an object's data matches the default data in the class definition, the XML representation does not spell out the object values but simply indicates that the object has the default data. An interpreter must then know what the default values are for the current class definition, which is not appropriate when sharing data between applications written in different development environments. This schema is convenient for use within LabVIEW and the team does use it in these circumstances.

For more general XML formatting the DCT team instead uses JKI Software's EasyXML toolkit, which uses a schema modeled on SimpleXML. The DCT team developed customized solutions for two situations this toolkit does not address:

2.3.4.1.1 Values with units

The DCT team uses LabVIEW's ability to designate units for numerical values in some applications to ensure unit consistency. When EasyXML's Easy Generate XML function writes a value with a unit to XML, it always writes the value on the wire, which is always in the base unit. The DCT team wrote a set of methods to write the value in the desired unit.

2.3.4.1.2 Objects

The Easy Generate XML function generates an error if the data input is a LabVIEW object (or other reference object). The DCT team wrote methods to parse data into its constituent parts. When a part is not a reference object, the parser calls the Easy Generate XML function to create the relevant piece of XML.

This task was particularly difficult because the type descriptors for LabVIEW objects are not available at run-time to nonNI LabVIEW developers. This situation means it is not possible to write an XML framework tool using the methods normally available in available versions of LabVIEW. The DCT team implemented a consistent but specialized set of code that necessarily deals with each situation uniquely. It is our hope that NI will make object type descriptors available at run-time to external developers, or, better yet, will implement an XML schema that will allow for simple data sharing with third-party applications.

3. OBJECT-ORIENTED ANALYSIS AND DESIGN

3.1 LabVIEW Object-Oriented programming

National Instruments first supported a native by-value object implementation for desktop applications in 2006. In LabVIEW 2009 NI extended this to the real-time and FPGA platforms, as well as introducing a by-reference implementation option.

The DCT team uses by-value classes extensively both in Windows applications and now in real-time applications. (Our first use of classes on a real-time platform was for the primary mirror position controller.) The team has not yet encountered a need to use the by-reference implementation.

This effort has realized benefits as well:

1. Meaningful data encapsulation. In the example, since ‘m1pData’ encapsulates the Model (which is fairly complex), we can hide the details of the model in single wire and execute operations on the model as needed (see Figure 14).
2. Implementation of Object-Oriented design patterns (particularly the State Pattern and Command Pattern)
3. Code reuse (XML Configuration Files, ComponentData)

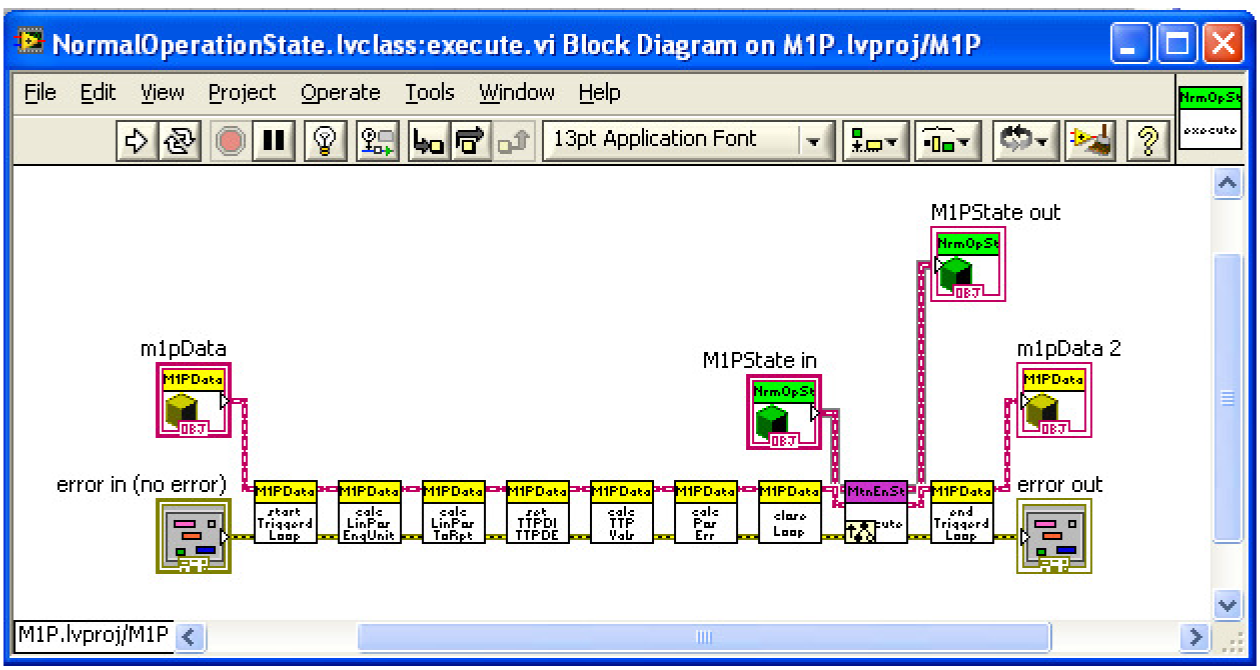

Figure 14. Encapsulation in m1pData

3.2 Design patterns

The DCT software uses Object-Oriented design patterns[2] where appropriate.

3.2.1 Command pattern

We have already presented an example of the Command Pattern above.

3.2.2 State pattern

Since DCT software components are mostly controllers that have clearly definable behavior under different circumstances, using a statemachine-based design is clearly beneficial. The team implements statemachines using the State Pattern design pattern, in which each state is a first-class object with a single execute method. The execute method has a single input and output that is the system model. An execute method generalizes the parent method for override or extension. For instance, we have already shown above a statemachine diagram for the component-level summary states. Each state maps to a first-class object, as we show in the following class diagram (see Figure 15).

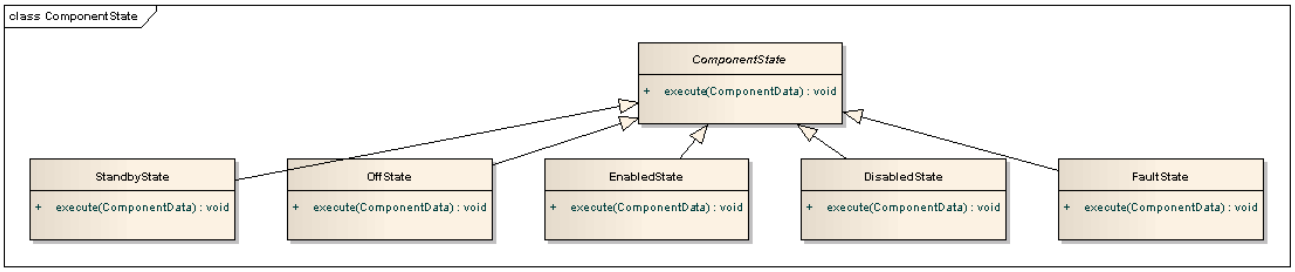

Figure 15. ComponentState class diagram.

Note that we can (and usually do!) have multiple levels in the hierarchy, which allows us to specify common behaviors in superstates.

3.2.3 Model/view/controller

Model/View/Controller (MVC) does not appear as one of the 23 cataloged patterns in Design Patterns, but the authors do discuss MVC as an example that uses multiple design patterns.[2] A common instantiation of MVC in a DCT component is as follows:

1. The Controller application runs in real-time on a compactRIO. It implements a statemachine that oversees the functionality of the core component. The Controller changes state based on data it receives via external messages. In particular, the Controller subscribes to specified shared variables. Publishers to these shared variables may include the user interface for the component (the View) or an external component. In turn, the Controller for the component we are considering publishes information on its current state to a different set of shared variables. Subscribers to this information again include the View for this or other components, as well as Controllers associated with other components. The Controller can continue operating if it does not receive new instructions.

2. The View is a user interface running on a desktop. Since the Controller does not depend on the View in order to operate, the user can close or open the View at any time. User actions on the View result in the generation of signals to the Controller. The View subscribes to and displays state information from the Controller.

3. The Model is a representation (as an object) of the system state. The execute method for each state in the Controller performs actions that may result in changes to the Model (i.e., changes to the model object state). In real-time applications, for example, each execute method completes within the required loop time and returns the current Model for use in the ensuing state.

4. SOFTWARE DEVELOPMENT ENVIRONMENT

4.1 LabVIEW

The DCT software development team, with a nod to the Southern Astrophysical Research (SOAR) Telescope,[3,4,5,6] the Hobby-Eberly Telescope,[7] and the South African Large Telescope (SALT), which all use LabVIEW to some extent, has adopted National Instruments LabVIEW as the primary, but not exclusive, development environment. Some reasons in support of this decision include:

1. The team finds that LabVIEW's graphical development environment enhances code development as well as code comprehension by (knowledgeable) readers. In particular the use of a graphical coding environment reduces the representational gap between model and code. We have found that data encapsulation in objects further enriches this advantage.

2. LabVIEW's dataflow paradigm makes the implementation of parallel operations quite simple.

3. LabVIEW offers a convenient programming paradigm that is applicable across the application space for control applications, including user interfaces for control and for the display of state information, desktop applications, real-time applications, and hardware interface (including FPGA) applications.

4.2 Others

C++ and Java applications also exist in the system. Applications written in one language interact with direct calls or through the messaging system described above.

5. XML CONFIGURATION FILES EXAMPLE

The team has developed a configuration handler that utilizes certain architectural features, including the use of LabVIEW objects, design patterns (Command Pattern), and the shared variable publish-subscribe communication paradigm. We have already shown the essentials of the ConfigurationCommand class hierarchy, communication, and dynamic dispatch implementation above.

The resulting implementation has the following advantages:

1. Code reuse: To create a new configuration files application a developer simply extends existing classes.

2. Native schema: The application uses LabVIEW's internal XML read and write methods, rather than custom methods currently in use to structure data for use in .ini files.

3. Data definitions: The application creates a unique file for each data definition with the name of the data definition. Unlike some existing methods commonly used in LabVIEW with .ini files, only the exact data desired appears in the XML file, and the ordering in the file is always precise.

See Figure 16 for an example of the XML configuration files user interface.

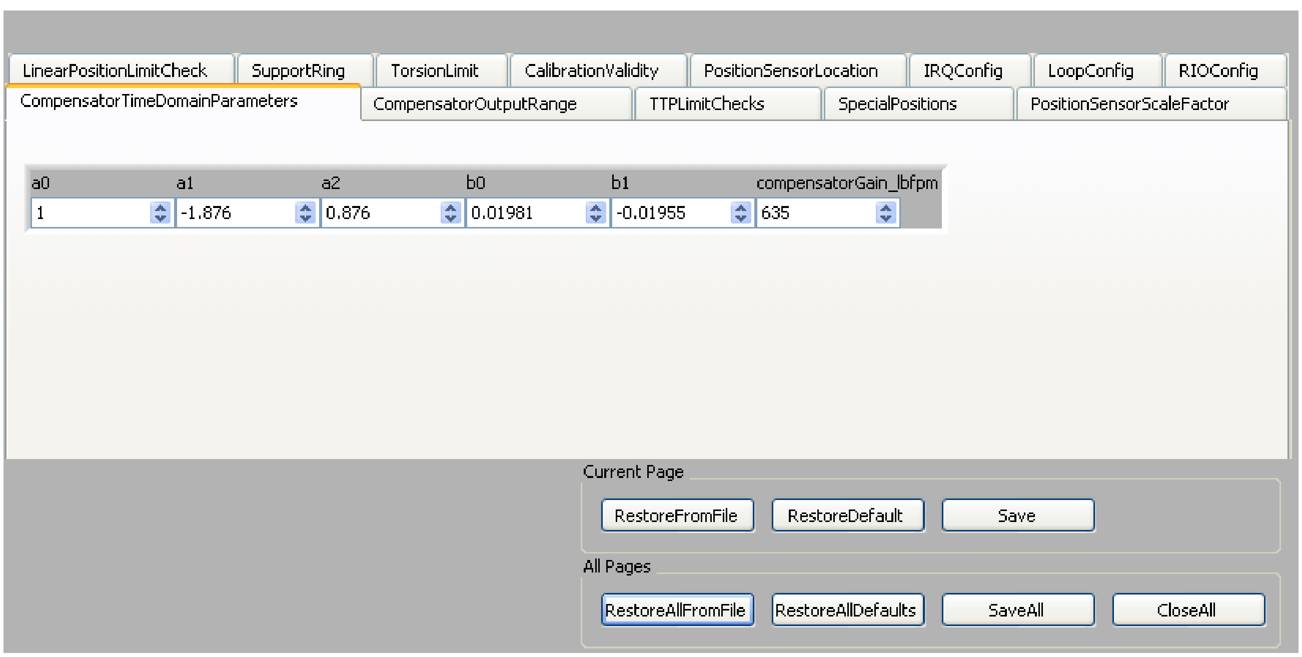

Figure 16. Example configuration screen.

6. SOFTWARE ENGINEERING TOOLS AND TECHNOLOGIES

We mention some of the more helpful tools and technologies our team has incorporated in our iterative software engineering process.

6.1 Enterprise Architect

The team uses Sparx Systems Enterprise Architect for modeling both at the system level and at the software design level.

6.1.1 UML

Enterprise Architect is a computer-aided software engineering (CASE) tool for the Unified Modeling Language (UML). We have found the UML useful not only for visualizing design ideas but also for documenting Object-Oriented designs and interfaces.

6.1.2 SysML

The team also utilizes the Systems Modeling Language (SysML) as well to capture and provide traceability between use cases, requirements, test cases, and designs.

6.2 JIRA

The team relies on Atlassian JIRA for issue tracking. JIRA's web-based interface makes it easily accessible and provides a shared interface to follow issue progress.

6.3 Greenhopper

The Greenhopper JIRA plug-in facilitates the team's iterative planning and progress tracking efforts.

6.4 Enterprise Tester

The team uses Enterprise Tester, developed by Catch Limited, to facilitate the execution of test scripts, record test results, and generate test reports. Enterprise Tester imports requirements and test procedures created in Enterprise Architect. If a test fails, an Enterprise Tester user can generate a JIRA issue from Enterprise Tester.

6.5 Subversion

The team relies on Subversion for version control. We check in most types of files manually using the TortoiseSVN Windows client. We use Enterprise Architect's user interface to check in models (XMI files); Enterprise Architect itself uses a CollabNet Subversion client.

7. CONCLUSION

We have presented the key architectural concepts of the DCT software's component-oriented design, discussed the implementation of the publish-subscribe messaging system required to support this architecture, detailed the use of Object-Oriented technologies in LabVIEW in desktop and real-time applications, including its use in our own XML configuration files tool, and described engineering tools that support the development process.

REFERENCES

[1] Schumacher, G. and Delgado, F., "The Large Synoptic Survey Telescope Middleware Messaging System," Proc. SPIE 7019 (2008).
[2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., [Design Patterns: Elements of Reusable Object-Oriented Software], Addison-Wesley, Boston, (1995).
[3] Ashe, M. and Schumacher, G., "SOAR telescope system: a rapid prototype and development in LabVIEW," Proc. SPIE 4009, 48-60 (2000).
[4] Ashe, M., Schumacher, G., and Sebring, T., "SOAR Control Systems Operation: OCS & TCS," Proc. SPIE 4848, 294-303 (2002).
[5] Ashe, M., "ArcVIEW: a LabVIEW-based astronomical instrument control system," Proc. SPIE 4848, 508-518 (2002).
[6] Schumacher, G., Heathcote,S., and Krabbendam, V., "SOAR TCS: from implementation to operation," Proc. SPIE 5496, 32-37 (2004).
[7] Hall, D., Ly, W., and Howard, R., "Software development for the Hobby-Eberly Telescope's Segment Alignment Maintenance System using LabVIEW," Proc. SPIE 4848, 239-250 (2002).