Discovery Channel Telescope software development overview

Paul J. Lotz*[a], Daniel Greenspan[a], Ryan Godwin[a], Philip Taylor[b]
[a]Lowell Observatory, 1400 W. Mars Hill Road, Flagstaff, AZ 86001;
[b]Observatory Sciences Ltd., Williams James House, Cowley Road, Cambridge CB4 0WX UK

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. We present an overview of the current status of the project software effort, including the iterative development process (including planning, requirements management and traceability, design, code, test, issue tracking, and version control), our experience with management and design techniques and tools the team uses that support the effort, key features of the component-based architectural design, and implementation examples that leverage new LabVIEW-based technologies.

Keywords: DCT, Lowell Observatory, software engineering process, requirements management, Enterprise Architect, UML, LabVIEW, shared variable, telescope software, component, cRIO

1. INTRODUCTION

After describing the system components and architecture, we summarize the current state of the development effort and describe the key technologies used on the project, then conclude with a look at the future.

2. SYSTEM OVERVIEW

The DCT team has allocated system behaviors at the top level to the following stand-alone software components of these components further allocate behavior to subcomponents. The components generally fall into two groups: instruments and telescope systems. Figure 1 shows the DCT software high-level components.

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

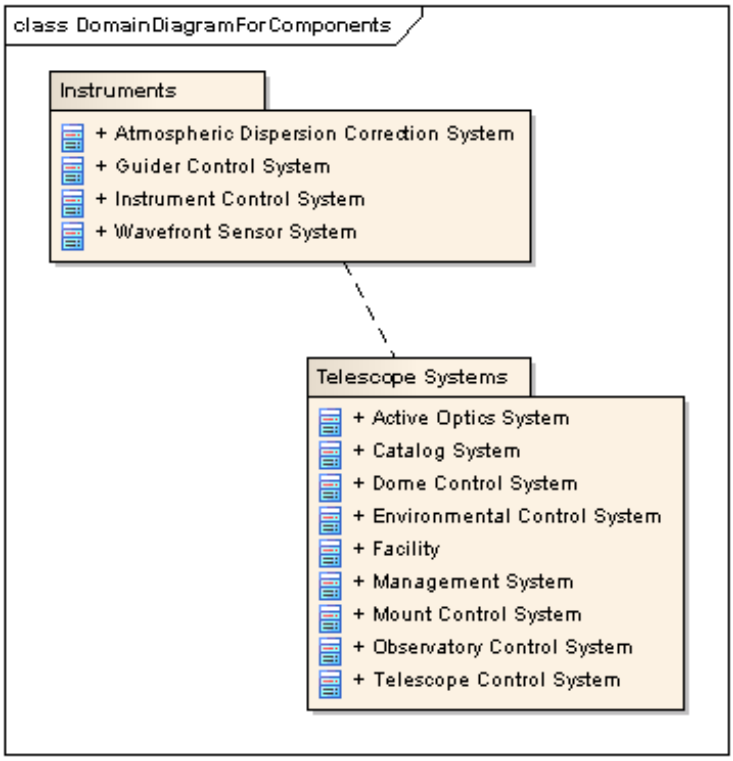

Figure 1. DCT software high-level components.

The telescope system components run as independent applications and publish and subscribe to data messages. The Observatory Control System exercises a loose supervisory role. Figure 2 shows the components of the telescope system.

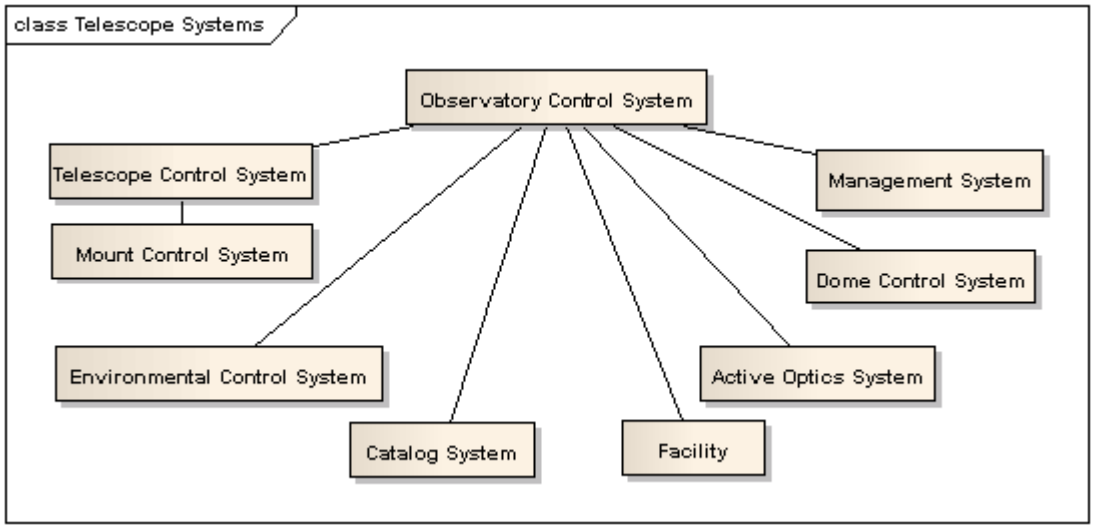

Figure 2. Components of the telescope system.

2.1 Mount control system

This subsystem controls the motion of the azimuth, elevation, and Cassegrain rotator axes.

2.2 Telescope control system

This subsystem streams time-stamped position and velocity demands to the mount control system. In addition, it sends expected target positions (in an ideal plane) to the guider control system and wavefront sensor, as well as atmospheric dispersion demands to the corrector. This system incorporates guide target measured errors, secondary mirror positions, weather data, and dispersion corrector optical deviations into the pointing model to achieve the demanded pointing. Target demands arise from the user interface of this component or from external components.

2.3 Observatory control system

This component provides a supervisory role, monitoring system states and coordinating operations (when required) between components. It will also provide a system-level user interface.

2.4 Guider control system

The guider provides feedback (guide error) based on the difference between the actual and predicted positions of the guide target. It also measures the guide target image spot size.

2.5 Wavefront sensor

The wavefront sensor measures image quality and publishes fit residuals as bending mode coefficients (for primary mirror corrections) and a limited set of Zernike coefficients (for primary or secondary mirror corrections, depending on the optical configuration), as well as information required to calculate derived image quality.

2.6 Instrument control system

This component can send target demands to coordinate an observation. Most additional functionalities are highly dependent on the particular instrument and are outside the scope of the project.

2.7 Atmospheric dispersion corrector

The corrector compensates for the dispersion as indicated by the telescope control system, and reports the resulting pointing offset (if any).

2.8 Dome control system

This component controls the dome azimuth rotation and shutter open and close operations. It can track the demanded mount positions, move to a user-specified position, or move to a special position to support calibration operations.

2.9 Active optics system

This system optimizes image quality. Correction factors will derive from calibration tables based on measured image quality at a set of elevation values and temperatures and from wavefront sensor feedback.

When the telescope is operating in its Ritchey-Chrétien configuration, the active optics system controls the figure of the primary mirror and the tip, tilt, and piston of the secondary mirror. When the telescope is operating in its prime focus configuration, the active optics system controls the figure, tip, tilt, and piston of the primary mirror.

The active optics system includes the following controllers for the primary mirror:

1. The system controls 36 lateral supports on a single pneumatic circuit via a pressure regulator. Feedback to the controller derives from measurements from force transducers associated with three passive tangent definers.
2. The system controls the tip, tilt, and piston of the primary mirror. Feedback comes from four position sensors located around the outer diameter of the mirror and the output of the compensator is a suite of forces that forms the basis of setpoints for the axial support force loop.

3. The system controls 120 axial support stepper motors. Each stepper motor has its own controller. In line with each actuator is a force transducer that provides feedback to the control loop.

In addition, the active optics system includes the following controllers for the secondary mirror:

1. The system maintains zero force on the volume between the secondary and the cell structure by controlling the state of four solenoid valves, two that gate vacuum and two that gate pressure, based on feedback from three force transducers and a differential pressure sensor.
2. The system performs closed-loop position control of the secondary mirror using three axial actuators with feedback from position sensors.

2.10 Environmental control system

The team has assigned functionalities related to environmental monitoring and control to this component. These functionalities include measuring and controlling the ambient temperature at the observing level (based on feedback from a weather station and from temperature sensors located in the dome, the system influences the temperature by operating ventilation fans and opening and closing ventilation doors). This system also controls the dome lamps and calibration screen lamps.

2.11 Facility

This system will provide feedback on the state of facility infrastructure systems.

2.12 Catalog

The Catalog will support the selection of telescope targets. It will offer features such as bounds-checking and some search capabilities.

2.13 Management

This component will collect data on system utilization and provide basic facilities for analysis of this data.

3. KEY ARCHITECTURAL CONCEPTS

The system consists of loosely interacting components implemented as far as practicable with an Object-Oriented design.

3.1 Component design

Allocating functionality to stand-alone components as we have described above is a key feature of the software design. Each software component responds to external stimuli (data that arrives as signals) depending on its current state. This is quite similar to the concept of Concurrent Components Communicating Anonymously (C^3A) the Large Synoptic Survey Telescope LSST (team) has previously described.[1]

3.1.1 Communications

The system supports asynchronous messaging largely via publish-subscribe protocols, but it also supports a basic TCP/IP message passing protocol (using ASCII commands) in the case of communications between the telescope control system and the mount control system.

3.1.1.1 Publish-subscribe protocol

Systems subscribe to signals bearing relevant data and publish data to a different set of signals. A particular signal can support multiple subscribers. Each client interacts only with the message server.

The system has two types of message servers. Communications between and within LabVIEW-based components utilize the National Instruments shared variable engine. Communications outside this realm rely on an ActiveMQ message broker (an implementation of the Java Message Service API).

3.1.1.2 Message content

The message content takes several forms:

1. ASCII strings for messages between the telescope control system and the mount control system.
2. Native LabVIEW types where practical.
3. LabVIEW objects flattened to strings where practical between LabVIEW-based components.
4. An XML representation of data in the form of basic types or objects for communication between components written in different software languages. For each relevant software language (LabVIEW and Java currently) we require an XML parser.

3.1.2 Object-Oriented analysis and design

For all the common reasons, most notably to provide for encapsulation and code reuse via generalization, we use Object-Oriented designs. The software team uses LabVIEW's native Object-Oriented programming capabilities first released in the language in 2006 and extended to the real-time platform in 2009.

4. PROGRESS HIGHLIGHTS

We summarize progress to date on key system components.

4.1 Mount control system

General Dynamics SATCOM Technologies has implemented a version of the software for this component. GDST customized software currently in use in other applications to create the current version.

4.2 Telescope control system

Observatory Sciences Limited, under contract to Lowell Observatory, has delivered a version of the telescope control system. This component utilizes the TCSpk pointing kernel licensed by Tpoint Software.[2]

4.3 Observatory control system

The team has developed a C++/LabVIEW ActiveMQ client and a LabVIEW-based XML interpreter that reside here.

4.4 Guider control system and wavefront sensor

The Lowell Instrument Group is currently developing software for the initial guider control system and wavefront sensor.

4.5 Instrument control system

The Lowell Instrument Group is developing software for common services and for specific instruments.

4.6 Dome control system

The team has completed software for the dome.

4.7 Active optics system

The team has completed the primary mirror control portions of the software for the active optics system. Work on the secondary mirror is partially complete and implementation of the lookup tables and corrective factor calculations is ongoing.

4.7.1 Testing

The team completed testing for the primary mirror active optics system hardware (120 axial supports and 36 lateral supports) in early 2010.

Tests for each axial actuator verified compliance with requirements for stiffness, hysteresis, stroke, electrical limit functionality, breakaway force limits and functionality, and closed loop performance. The team developed a data analysis tool that retrieved data logged to the Citadel historical database, displayed the data, and performed a series of

repeatable tests. Analyses of early tests revealed issues with nonlinear force regions, out of range breakaway forces, and control loop performance with the existing plant. The DCT design team resolved all these issues by applying a combination of electrical and mechanical solutions. Subsequent testing revealed compliance with specifications.

Tests for lateral supports examined freedom of motion of the diaphragms, leakage in the diaphragm and pneumatics, hysteresis, and rated load.

4.8 Environmental control system

The team has implemented software that publishes data from a weather station. Additional software controls the ventilation fans and ventilation doors, as well as the dome lights.

5. TECHNOLOGIES

We describe some key implementation and software engineering technologies in use on the project.

5.1 Software infrastructure

We describe some of the key technologies used in the project. For more details, see our other paper.[3]

5.1.1 LabVIEW

The primary development environment the team uses is National Instruments LabVIEW, a graphical programming language previously used on the Sourthern Astrophyiscal Research (SOAR) Telescope,[4,5,6,7] the Hobby-Eberly Telescope,[8] and the South African Large Telescope (SALT). LabVIEW allows the team to apply a single strategy for development on FPGAs, real-time, and PC platforms. The team's subjective experience is that graphical programming lends itself well to comprehension. LabVIEW's dataflow paradigm simplifies parallel programming.

As noted above, LabVIEW now supports objects natively on PCs, real-time, and FPGA platforms.

Many of DCT's applications require real-time performance. We deploy the real-time controllers on a compact Reconfigurable Input Output (cRIO) platform from National Instruments. Networked real-time FIFO-enabled shared variables provide a means of data communication between the controller and the outside world (including the view for the application, which runs on a PC). The use of Ethernet-based communication eliminates the need for any data acquisition hardware inside any PCs, and allows access to data anywhere on the network.

5.1.2 Networked shared variable

The National Instruments LabVIEW shared variable provides a means of communicating using a publish-subscribe protocol native to LabVIEW. For each application we deploy a shared variable engine (server) on a PC. With the addition of the Datalogging and Supervisory Control (DSC) module we are able to utilize features such as logging to a Citadel historical database. We host the server on a Windows machine rather than a cRIO, where applicable, since National Instruments only supports the logging features of the Datalogging and Supervisory Control (DSC) module on a Windows platform. The most local PC hosts the server (rather than having one large server) to permit each system to operate stand-alone. For data analysis we can merge databases later.

The type of a shared variable generally can be a native LabVIEW type or a user-defined type. The list of available types, however, does not include LabVIEW objects.

Nonetheless, the team successfully implemented a version of the Object-Oriented Command Pattern[9] utilizing string-typed shared variables.

The sending application flattens the object to write to a string (see Figure 3).

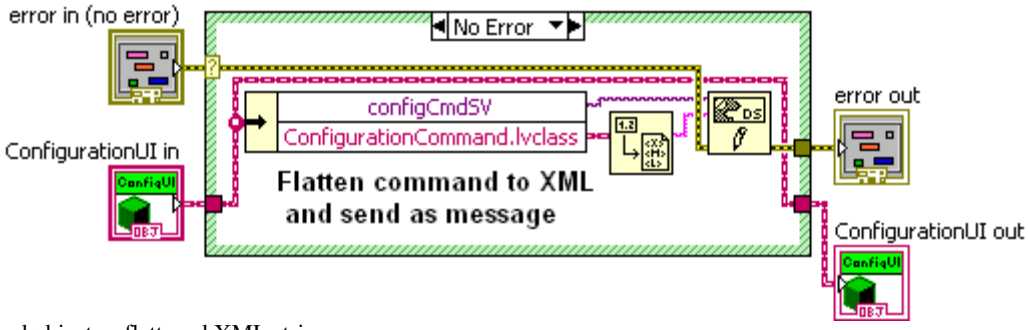

Figure 3. Send object as flattened XML string.

The receiving application unflattens the string to the parent command type (see Figure 4).

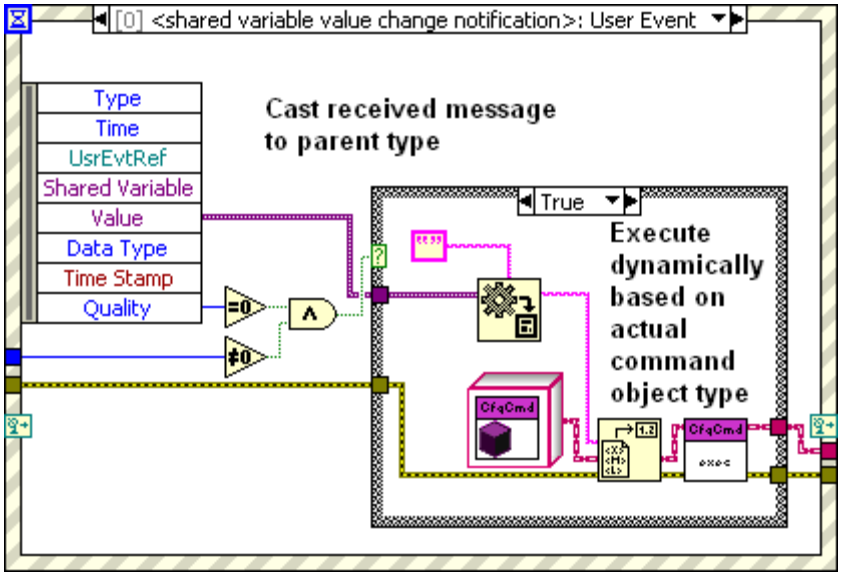

Figure 4. Unflatten command from XML and execute.

Real-time FIFO-enabled shared variables have a more limited list of available types. (Most notably, LabVIEW needs to know the message size at compile time.) As a consequence, we limit shared variable communication to real-time systems to use a more restricted set of types.

5.1.3 Java messaging service

For messaging between nonLabVIEW components and between these components and LabVIEW we generally use an Apache ActiveMQ message broker (an implementation of the Java Message Service API). The team developed a C++ client based on Apache's existing CMS client. When the C++ client receives a message via a callback from the message broker it generates a LabVIEW user event, which allows event-based message handling in the LabVIEW application as well.

5.1.4 Historical database

As noted above, applications log shared variable data to a Citadel historical database hosted on a Windows PC. This off-the-shelf solution has eliminated the task for the team to develop its own logging service. The LabVIEW DSC module has methods for querying the database. We found these to be sufficient, although it is possible to develop one's own methods to query the database using SQL.

Methods exist to archive or merge the databases.

5.1.5 XML

We use XML in the following ways:

5.1.5.1 XML for communication between development environments

XML provides us the ability to describe data in a common way. We leverage this capability to share data (including object data) between platforms.

The team uses an XML implementation based on SimpleXML. Since this capability is not native to LabVIEW, the team had to create its own parser to handle object data. The current implementation is necessarily nonoptimal in design (LabVIEW does not provide access to an object's private data at run-time) but functional.

Note that LabVIEW does have a native XML schema but we found this parser unsuitable for the purpose of sharing data between applications written in different development environments for the following reasons:

1. LabVIEW's native XML parser uses a somewhat unique schema that does not easily port to other development environments.
2. When a LabVIEW object has default data LabVIEW's XML schema merely represents the data as default, rather than fully expressing the data. Consequently a receiving application requires access to LabVIEW's definition of the object for interpretation.

5.1.5.2 XML for storing configuration information

The limitations of LabVIEW's native parser are not problematic when used within LabVIEW.

We have leveraged LabVIEW's native XML capabilities to build an XML-based configuration handler. The resulting application has the following benefits:

1. The implementation uses the Object-Oriented Command Pattern. The core functionality exists in a common object library.
2. Implementing a configuration file handler for a new component requires little more than defining the particular data elements to display and their allowable input ranges.
3. Data from each tab stores in a unique human-readable file, allowing for easy maintenance.
4. Data from irrelevant controls does not appear in the configuration files.

5.1.6 Component

The team has implemented a Component class and related classes that include functionality common to the various components. This functionality includes:

1. Component-level state machine
2. Timestamp information
3. A loop timer
4. Error information
5. A handler for interrupts (for applications deployed on cRIO systems)
6. RIO information (for applications deployed on cRIO systems)
7. A handler for common signals (i.e., for handling errors, state change signals such as start, standby, exit)
8. Configuration path information

5.2 Software engineering tools

The DCT team relies on a set of tools and technologies to optimize the development process.

5.2.1 Requirements management

Most requirements are currently in text-based documents or in a requirements database management tool (DOORS). The team has recently begun representing requirements as elements in a single model using SysML in Enterprise Architect.

5.2.2 Version control

The team uses Subversion for version control. We use the TortoiseSVN client for most purposes, but Enterprise Architect interfaces to the Collabnet command line client.

5.2.3 Schedule planning

The team utilizes Microsoft Project for some aspects of planning.

Recently the team has adopted the use of the Atlassian Greenhopper JIRA plug-in to help us monitor our schedule more closely. This we see has the following advantages:

1. Integration with JIRA
2. Ability to log work
3. Simple interface
4. Ability to track iteration progress and burn-down rate

5.2.4 Modeling

The team uses Enterprise Architect for UML (and SysML) modeling applied to the following tasks:

1. Provide illustrations to facilitate discussions of possible implementations
2. Document designs and the resulting implementations
3. Generate code. (We have done this for Java classes. LabVIEW does not support code generation from Enterprise Architect, although a different tool Endevo UML Modeller, does support LabVIEW code generation.)

5.2.5 Testing

The team has begun using Enterprise Tester to execute test procedures and to store and report on test results. The team creates requirements and develops a test plan (using the structured scenario feature) in SysML in Enterprise Architect. The team then imports the requirements and test procedures into Enterprise Tester. Enterprise Tester facilitates the execution of test procedures via execution sets, linking test results to requirements to complete the traceability path. Enterprise Tester generates a verification matrix that shows the current status of each requirement, and it can link to JIRA to generate issues when tests fail.

5.2.6 Issue tracking

The team uses Atlassian JIRA for issue tracking.

6. REMAINING QUESTIONS

Most of the remaining technical questions the team faces center on optimizing user interfaces. For instance, we use plug-ins currently but we can certainly improve how we do this. Unfortunately, creating dockable windows is not a native LabVIEW feature. We need to improve graphical user interface (GUI) appearance stability across machines and monitors (we are almost there). Most important, we need to apply appropriate design principles for visual flow.

REFERENCES

[1] Schumacher, G. and Delgado, F., "The Large Synoptic Survey Telescope Middleware Messaging System," Proc. SPIE 7019 (2008).
[2] Wallace, P., "A rigorous algorithm for telescope pointing," Proc. SPIE 4848, 125-136 (2002).
[3] Lotz, P., "Discovery Channel Telescope software key technologies," Proc. SPIE 7740 (2010).
[4] Ashe, M. and Schumacher, G., "SOAR telescope system: a rapid prototype and development in LabVIEW," Proc. SPIE 4009, 48-60 (2000).
[5] Ashe, M., Schumacher, G., and Sebring, T., "SOAR Control Systems Operation: OCS &TCS," Proc. SPIE 4848, 294-303 (2002).
[6] Ashe, M., "ArcVIEW: a LabVIEW-based astronomical instrument control system," Proc. SPIE 4848, 508-518 (2002).
[7] Schumacher, G., Heathcote,S., and Krabbendam, V., "SOAR TCS: from implementation to operation," Proc. SPIE 5496, 32-37 (2004).
[8] 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).
[9] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., [Design Patterns: Elements of Reusable Object-Oriented Software], Addison-Wesley, Boston, (1995).