TRENDS IN SOFTWARE FOR LARGE ASTRONOMY PROJECTS

G.Chiozzi*, A.Wallander, ESO, Garching, Germany
K.Gillies, Gemini Observatory, La Serena, Chile
B.Goodrich, S.Wampler, National Solar Observatory, Tucson, AZ, USA
J.Johnson, K.McCann, W.M. Keck Observatory, Kamuela, Hawaii, USA
G.Schumacher, National Optical Astronomy Observatories, La Serena, Chile
D.Silva, AURA/Thirty Meter Telescope, Pasadena/CA, USA

Abstract

Current 8-10m ground-based telescopes require complex real-time control systems that are large, distributed, fault tolerant, integrated and heterogeneous. New challenges on the horizon (laser guide star, adaptive optics, 30–40m class telescopes) will bring increased complexity, where requirements cannot be cleanly isolated due to component coupling within the control and acquisition chain. Moreover, the high cost of observing time imposes challenging requirements on system reliability and observing efficiency. New, more complex facilities are not simply scaled up versions of current facilities – paradigm changes may be required.

Here, responses by the astronomical community to these new software challenges are discussed. Our focus is the evolution of control system architecture and software infrastructure from the current generation of facilities to the next. Although we focus on control systems, clearly this is just one of the several subsystems integrated within the entire observatory end-to-end operation.

INTRODUCTION

Over the last 10 years, software engineers at the current generation of 8–10m telescopes have adapted to significant changes in requirements and technologies relative to their predecessor 3–4m class telescopes[1]. With segmented and large thin mirrors came the need for active mirror control. Even more significant was the introduction of adaptive optics, laser guide stars and optical interferometry. Along with increased control model complexity came higher demands on bandwidth, data storage, and data processing. The primary challenge has been to incorporate such new technology and requirements while achieving acceptable system reliability and observational efficiency.

At present, a new generation of facilities is on the horizon, from a 4m class advanced technology solar telescope to an 8m class wide-field survey telescope to one or more 30 – 40m class telescopes. Each of these new facilities inherits the software challenges of the last generation while adding new complexity of their own.

In this paper we will analyze and compare various aspects of current telescope control systems, with an emphasis on temporal evolution. From the many areas that could be discussed, we focus on challenges and evolution in the areas of system architecture, development methodology, and technological implementation.

*gchiozzi@eso.org

ARCHITECTURE

All major operational telescopes use a familiar three-tier hierarchical architecture:

1. High-level coordination systems
2. Low-level real-time control computers (LCUs, local control units).
3. Devices with a limited degree of intelligence directly connected to hardware

Each system is mostly comprised of fairly independent subsystems. Sub-system interaction is typically limited to slow correction offloading from faster to slower subsystems within a wider operational range.

Wave front control using adaptive optics systems and optical interferometry dramatically change this simple fast-to-slow offloading paradigm, since they require real-time coordination of the feedback among many sensors distributed throughout one or more telescopes. System architectures must now deal with real-time devices with significant physical separations that must be coordinated in real-time. In the past it was sufficient to use time tagged commands sent in advance in a few special situations, but general, system-wide synchronization and latency minimization are now required.

Techniques can be used to localize command & control, but this is insufficient when the devices to be controlled are far away from one another and fast, deterministic synchronization is needed. In that case, control processors need to be close to the mechanisms and connected to other controllers by a deterministic network or industrial bus. In many cases, observatories have evolved into systems of systems, where different lineages of subsystems must function together in a heterogeneous fashion, possibly sharing infrastructure and needing to share common models and communication solutions.

In parallel to these developments, LCUs once necessary to perform both soft and hard real-time tasks and used as general purpose I/O units to connect to field HW (sensors and actuators) are being replaced. Nowadays, soft real-time tasks can be performed directly by the coordination workstations while the HW devices are much more autonomous/intelligent and can normally be connected

directly to an Ethernet network or other standard communication buses.

FRAMEWORKS

The value of providing a uniform software framework to simplify the development and maintenance of complex control systems is obvious[2]. EPICS, adopted by Keck and Gemini, has been one of the first solutions thought to facilitate the creation of distributed soft real-time control systems. EPICS deals with many of the technical aspects of the system providing a common programming environment and common services.

Recently, this concept has been more formally defined and implemented through the separation of the *functional* and the *technical* aspects of large, distributed software systems. The so-called technical framework defines the software elements that are independent of task-specific components, which in turn are layered on top of the technical framework.

In the past, the technical framework was limited to common libraries and sets of guidelines on how to access and use those libraries. The advent of object-oriented programming techniques has made it possible to extend the scope of the technical framework, effectively providing automatic implementation of many operations previously covered only by guidelines. This allows software developers to focus more on the task-specific implementation.

Observatories such as ALMA and ATST as well as newer subsystems of current observatories have adopted a *component-container model*[3]. SW applications consist of independent *components* that communicate with each other using a number of integrated services. Typical technical aspects of system integration, like distribution, deployment and location of other *components*, are hidden to the developers. All these technical issues are the responsibility of *containers* provided by the framework: they are the ecosystem where *components* run and find convenient access to all services they might need.

A component-based architecture is particularly useful in distributed and heterogeneous systems typical for our observatories, since it facilitates the practical partitioning of implementation into manageable units. However, for this strategy to be successful, the underlying technical infrastructure must ensure that the right information is delivered to the right place at the right time. The software layer involved in this information exchange is commonly known as *communication middleware* and supports the implementation of the connectivity and interoperability needs of the distributed applications.

However, it is important to isolate the application software from the underlying middleware, given the need to migrate to different technologies during the lifetime of a project. Examples of this software layer include the Keck Task Library (KTL)[4] and the ALMA Container Services[5]. These provide language independent solutions allowing a common API to be used for control coordination and monitoring of the observatory.

Common services adopted by the ALMA, ATST[6] and LSST[7] projects include:

- Connection: locate and connect to other applications in a distributed environment
- Event: high-performance asynchronous message handling
- Command: client/service communication for application control.
- Logging: telemetry capture as well as collection, recording, distribution and analyzes of messages.
- Persistent Store: hold system configuration information, calibration data, performance data, etc.
- Error Handling: monitor for improper behavior, report error conditions, and supports recovery operations

Through the use of standards (e.g. NVO, XML, CORBA), hopefully current technical frameworks in use at different observatories will integrate better and possibly even converge in the future. Observatories that share the same technical framework API will be able to easily share functional components. Such a convergence needs to be done carefully, however, as the nature of these technical frameworks is still in flux.

DEVELOPING METHODOLOGIES AND MODELING TECHNIQUES

All recent observatory projects tried to leverage existing, off-the-shelf software technology to lower overall development and maintenance costs. Of course, as development methodologies and modelling techniques have evolved, so has their use in observatory projects.

The problem of creating and describing a software design and verifying that it will satisfy project science requirements is extremely difficult. This is exacerbated by software development projects that are coupled to multi-year observatory design periods with multiple requirement and milestone reviews. We have seen that many traditional project management and engineering approaches do not work well with software.

The projects of the mid-90s were developed with the modeling tools and processes of that time. The imposed review structure lead to a waterfall development process to a greater or lesser degree. Object-oriented approaches were not widely adopted. For example, the control systems of Keck, Gemini, and VLT are largely C code, designed using structured programming methodologies such as Ward/Mellor.

New projects within these observatories, as well as new observatory projects such as ALMA, SOAR, and ATST, more fully embraced object-oriented programming and UML for design and modeling. The extent of UML use varies but is generally limited to what is needed to explain the software in a common sense approach. No project has yet embraced the complete UML development process.

Naturally, the most recent projects are experimenting with the most recent approaches. For example, both E-ELT and LSST are working with SysML (Systems Modeling Language) as a modeling approach that extends the ideas of UML to the entire system. SysML addresses two aspects that are particularly relevant for us:

- Requirement capture, management and traceability: the inability to handle changing requirements often leads to a crisis during the integration of sub-systems provided by different groups or vendors. Moreover, the sheer number of requirements is increasing quickly as a function of project complexity. Many of us share the conclusion that rigidly planned approaches simply do not work well in our domain and that new, more agile approaches are needed.
- Integration in a single, coherent, global system model, as seen from the point of view of different engineering disciplines (software, electronics, mechanics, optics and so on): software is the glue between all observatory sub-systems. In order to be able to coordinate the various parts, we need to understand the interfaces and the functional relationships.

Large-scale software development in the commercial world faces similar challenges, from which a variety of new tools and processes have arisen. Agile processes such as Scrum are highly relevant to the current problems of telescope and astronomy software development. Several of our projects (Gemini, ALMA) have successfully used iterative and agile ideas to put software releases in the hands of users quickly so they can provide feedback into subsequent releases.

The programmatic challenge is to integrate an agile software development approach with the mandated waterfall approach of funding agencies. In principle, this can be accomplished through education and positive experiences. It is still too early to measure the programmatic success of this approach.

TECHNICAL ASPECTS

Hardware platforms

It is common to assert that hardware choice is driven by system architecture, but pragmatically it is often available hardware capability that dictates system architecture. In most operating observatories:

- High-level coordination is performed by general purpose workstations.
- Real-time tasks handled by local control units (LCUs), often VME-based.
- Devices (motor controllers, sensors, actuators) are directly attached to the VME computers using appropriate I/O interfaces.

Today many options are available in terms of cost and performance at all three levels.

On the upper level, high-end workstations have been replaced by standard personal computers, with a significant reduction in purchase cost. However, the tremendous pace in personal computer and component evolution makes staged purchases and upgrades extremely complicated because it is often impossible to find compatible HW after just a few months.

The soft real-time tasks once performed by the LCUs can now be handled by personal computers and (eventually) real-time extensions of the general purpose operating systems.

Intelligent devices, interconnected directly via Ethernet or an industrial bus like CAN, can now handle many localized hard real-time tasks. Furthermore, I/O data acquisition functions are being handled by distributed field point units rather than directly by an LCU . Very high performance real-time tasks (such as detector control or adaptive optics system control) are being off-loaded to DSP and FPGA based systems.

The implementation of fast distributed control loops to allow wave front control requires deterministic connections between the devices. At present, the most demanding systems use highly specialized solutions (for example both VLTI and Keck Interferometer use *reflective memory* technology). However, we foresee that generalized real-time and deterministic Ethernet or industrial buses will be widely used in observatories currently under development.

In specific situations, concentrated, raw computing power is needed. The ALMA interferometer requirement to correlate thousands of baselines per second is just one example. Such problems are very well suited for clusters and multi-core/CPU farms where latency and network delay are not a problem.

In parallel, solutions with high reliability and low lifecycle cost are highly desirable. A very interesting technology under evaluation for systems using large numbers of workstations (like the VLT) is virtualization. Using virtualization allows physical hardware to be de-coupled from the logical machine where the software is run. This has a number of advantages: deployment procedures are simplified, virtual hardware shields us from changes in real hardware simplifying upgrade strategies, and redundancy and hot swapping are handled by the virtualization platform as well as network reconfiguration.

Operating systems

Almost all control systems for observatories developed in the 1990s run high-level coordination software on top of a proprietary UNIX operating system, like Solaris or HP-UX. The real-time systems are based on a proprietary real-time operating system, commonly VxWorks.

In the last decade, open source solutions, especially Linux derivatives, have gained prominence at the high level. For example, the VLT has ported the complete control system to Linux and many projects are based on

this operating system. However, interest in Linux as a general solution has receded recently because total ownership cost relative to proprietary OSs has not declined as much as expected. Linux is now seen as one possibility among several that need to be evaluated for specific deployment decisions. Moreover, Solaris is re-emerging as a strong contender thanks to its x86 support and recent conversion to an open source product.

Interestingly, MS Windows does not play a major role in our observatory control systems, even though it is now widely used in industrial control applications and there is now a trend for contractors to supply Windows based solutions, with the *OLE for Process Control (OPC)* standard allowing different vendors to interoperate well. Only a few projects like the Discovery Channel Telescope are committed to the Windows platform.

In any case, the increasing role played in the upper application layers by Java and other high-level programming languages, which can enable effective independence from the underlying OS, can reduce the impact of OS choice.

In the arena of real-time operating systems, VxWorks and various flavors of Real-Time Linux play a major role, incarnating the usual advantages and disadvantages of the commercial versus free solutions. New (at least for us) solutions on the horizon include:

- Real-time Java and LabVIEW-RT solutions are extending their area of application and QNX is moving towards open source distributions.
- PLCs, with their proprietary simple OS and programming languages, are more often used to control commercial-off-the-shelf equipment
- FPGAs and DSPs are necessary for high performance requirements and bring their own specific development environments.

Programming languages

The main programming language used for the development of control applications between mid-80s and mid-90s was C. This was certainly true for the Keck, VLT, and Gemini observatories. Looking at the core telescope control functionalities (i.e. mount, dome, point and track, tip-tilt, chop, field rotation, mirror figures and alignment, interface and sequencing control), we can compare the application size (source lines of code) and programming languages actually used. Considering these projects were developed independently based on different frameworks/middleware the similarities are striking:

Language	Keck	VLT
C	251050	246738
C++	0	84400
Capfast	130116	0
Tcl/Tk	9408	81657
Others	118144	64136
Total	**508718**	**476931**

Later projects used C++, particularly in the high-level coordination role. For example, VLT development started with C and although C is the only language used in the LCUs, C++ is used in all high-level coordination applications. In ALMA C++ is used everywhere, down to the real-time applications.

However, we now see a clear decline in the usage of C++. On one side, Java is a much better object oriented language: it provides a huge amount of standard libraries, it is easier to learn and less error prone. For example, all ALMA high-level software is now implemented in Java and we have verified that porting C++ applications to Java (by the same expert C++ developer) resulted in a more reliable application. On the other side, C remains the favorite language for low-level real-time control. The structural complexity of these applications does not justify the introduction of an OO language, while at the same time it is easier to optimize algorithms in C. Not of least importance, good C developers are available in our observatories.

The glue between applications and high-level sequencing is normally implemented using an interpreted language, such as Tcl/Tk or Python. The emergency of new interpreted languages is quite rapid and there are always new options available. Although developing small applications is in general very easy, we all feel uncomfortable in allowing the usage of such languages for critical applications, because of the more limited possibilities of testing and verifying applications written with these languages.

Another language in use for control and monitoring applications is LabVIEW. This graphical programming development environment based on a dataflow paradigm is a radical departure from the sequential/deterministic style of text-based languages. SOAR and LSST project experience demonstrates that this paradigm adapts well to applications needed to control astronomical observations, in that the abstractions and structure of the software are closer to the real objects being manipulated.

Clearly, we use several programming languages at the same time, for different purposes. Hence, a major concern is application interoperability. Many languages allow us to link and call code from C/C++ libraries, but this is often not sufficient in highly distributed systems. Here appropriate middleware can provide the solution. ALMA for example has chosen CORBA to provide interoperability between applications and components written in a set of supported languages and other projects have done similar choices.

User Interface

User interface design and implementation continues to be a challenging area. Modern systems and users require more than just a proliferation of panels with simple widgets to display status values. Producing a good UI takes significant effort, which is often under-estimated and under-budgeted. We have concluded that good UI

design and implementation requires a skill set that most control engineers simply do not possess.

Looking across the observatories, Java and Tcl/Tk are the two most common technical solutions. Other solutions such as EPICS DM/EDD, Motif, Python/Tk or Qt are used to various degrees. In all cases the experience has been that the tools available are adequate for engineering panels but are not always sufficient to develop effective and user friendly end-user UIs.

In terms of technological choice, we see no clear winner and certainly not one that fits all solutions. Experience has shown that GUI builders do not provide good solutions, with virtually no support for the human engineering aspects and often produce code that is poor in performance and difficult to maintain. Such builders are most useful for the development of relatively simple UIs.

While modern tools and languages such as Python/Tk/Qt, LabVIEW or IDL promote rapid prototyping, very often these displays tend to be engineering centric and often bypass good development practices. Unfortunately, what are essentially prototype user interfaces often end up in operational environments, to the detriment of users and maintenance teams.

The human factor is also important in UI development. In general the UIs are growing more complex and require more attention with an emphasis on overall workflow and usability. While many requirements can be established early via use cases and end-user interviews, it is important to implement mock-ups and functional prototypes. The ability to rapid prototype UIs as part of a feedback loop with users is key to getting the human factor right. If this loop is too costly it can mean that the system ends up operating from overly technical engineering interfaces. Unfortunately, observatory projects often cannot afford to have an independent UI development team, experienced in working with end-users.

CHALLENGES OF NEW PROJECTS

Adaptive optics and laser guide star systems under development for existing facilities as well as the next generation of extremely large telescopes currently under development pose new challenges for control systems in both size and complexity. For example, the E-ELT incorporates a large 2.5 m deformable mirror with more than 5000 actuators and close to 1000 mirror segments controlled in tip, tilt and piston as well as overall shape. It is estimated that E-ELT will have more than 100 000 I/O points, a 10-fold increase relative to current 8-10m telescopes. A large number of control loops, some distributed, with sampling frequencies up to a few kHz have to work in unison. Stroke and closed loop bandwidth must be managed by multi-level off-load schemes. Further, the large number of actuators and sensors impose challenging requirements on fault detection, isolation and recovery.

At the same time, operational efficiency requirements have become more rigorous. For example, TMT has a requirement that target acquisition anywhere on the sky must be completed within five minutes. Ease of use and operational efficiency need to be considered at every stage of the software and hardware design and implementation cycle.

CONCLUSION

This paper summarizes an on-going analysis of control system evolution in astronomical observatories. From this analysis, we hope to share lessons learned and identify areas where greater cooperation across current operational observatories, as well as facilities under development, would be beneficial. The move towards open source paradigms and large international collaborations to share cost has made such collaboration more practical and affordable. In particular, we hope to be able to share technical architectural elements and infrastructure components more frequently, so that each observatory or project can focus on specific, science-driven needs.

ACKNOWLEDGMENTS

The authors would like to thank our many colleagues in the astronomical observatory community who have given so freely of their ideas and time as we have developed this paper and our various observatories and projects.

REFERENCES

[1] Wampler, S, and Goodrich, B, "Existing Telescope Controls", ATST Report RTP-0005, 2002

[2] K.Gillies, J.Dunn, D.Silva. “Defining common software for the Thirty Meter Telescope,” in Proc. of the SPIE Vol. 6274 - Astronomical Telescopes and Instrumentation, Orlando, Florida, USA, 2006.

[3] Chiozzi, G et. al., "Application development using the ALMA Common Software", Proc. of SPIE Vol. 6274 - Astronomical Telescopes and Instrumentation, Orlando, Florida, USA, 2006

[4] Lupton, W. F. 2000, “Keck Telescope Control System”, in ASP Conf. Ser., Vol. 216, Astronomical Data Analysis Software and Systems IX, eds. N. Manset, C. Veillet, D. Crabtree (San Francisco: ASP), 261

[5] Chiozzi, G et. al., "ALMA Common Software ACS Status and Developments", 10th ICALEPCS Int. Conf. on Accelerator & Large Experimental Physics Control Systems, Geneva 2005

[6] Wampler, S, "A middleware neutral common services infrastructure", 10th ICALEPCS Int. Conf. on Accelerator & Large Experimental Physics Control Systems, Geneva 2005.

[7] G. Schumacher, M. Warner, V. Kradbbendam, "LSST Control System", Proc of SPIE Vol 6274, 2006.