LSST OCS Status and Plans

Philip N. Daly[a], Germán Schumacher[b], Francisco Delgado[a], and Dave Mills[a]

[a]LSST, 950 N. Cherry Avenue, Tucson AZ 85719, USA
[b]LSST, Colina El Pino s/n, Casilla 603, La Serena, Chile

ABSTRACT

This paper reports on progress and plans for all meta-components of the Large Synoptic Survey Telescope (LSST) observatory control system (OCS). After an introduction to the scope of the OCS we discuss each meta-component in alphabetical order: application, engineering and facility database, maintenance, monitor, operator-remote, scheduler, sequencer, service abstraction layer and telemetry. We discuss these meta-components and their relationship with the overall control and operations strategy for the observatory. At the end of the paper, we review the timeline and planning for the delivery of these items.

Keywords: LSST, OCS, control software, application, engineering and facility database, maintenance, monitor, operator-remote, scheduler, sequencer, service abstraction layer, telemetry.

1. INTRODUCTION

The *Large Synoptic Survey Telescope* (LSST[1]) is an 8.4 metre ground-based optical survey telescope currently under construction on Cerro Pachón, Chile. The facility consists of the telescope and associated infrastructure, a 9.6 degrees^2 3.2 G × 16 bits/pixel camera,[2] a data management system[3] plus a suite of calibration instruments and a calibration telescope. The LSST is designed to survey 18,000 degrees^2 of the southern sky, in 6 broadband filters, with each region receiving ∼825 visits over the survey lifetime with a nominal cadence of 2×15 second observations per visit. The deep, fast, wide survey is intended to answer questions regarding topical science issues such as dark energy, cosmology and the contents (inventory) of the solar system plus provide serendipitous discovery through a large, curated database of observations and meta-data used by professional astronomers and citizen scientists alike.

To ensure that the survey can be carried out efficiently, an observatory control system (OCS) is being designed and constructed to meet the requirements, specifications and use cases derived from the science case. As the requirements document[4] explains:

> '*The functional requirements comprise the operations and behaviour that the OCS implements to accomplish the data collection capabilities of the observatory. The OCS works as the overall master control for the data collection functions of the observatory. The data collection capabilities comprise the capture of science data from the camera as well as the acquisition of calibration and telemetry data system-wide needed to analyze the system performance.*'

A SysML model[5] was developed of the OCS resulting in the principal components identified in table 1.

The OCS architecture, shown in figure 1, is loosely coupled with communications facilitated by the service abstraction layer (SAL[6]) of commands (using a command-action-response model), telemetry and events to other subsystems on the same bus. This bus utilizes the *OpenSplice* community edition of the data distribution service (DDS).

The OCS is the main meta-component in the data-centric control architecture of the LSST. Because of the middleware technology selected, and the distributed control paradigm, the OCS does not need to connect

Further author information: (Send correspondence to P.N.D.)
P.N.D.: E-mail: pdaly@lsst.org, Telephone: +1 520 318 8438
G.S.: E-mail: gschumacher@lsst.org, Telephone: +56 51 205347

Table 1. Main components of the observatory control system

Application	Business logic and rules
EFD	Engineering and facility database
Maintenance	Downtime manager
Monitor	Data monitoring and telemetry/event visualization
Operator-Remote	User interfaces
SAL	Service abstraction layer (infrastructure software)
Scheduler	Observation selection tool
Sequencer	Sequences observations
Telemetry	OCS specific meta-data

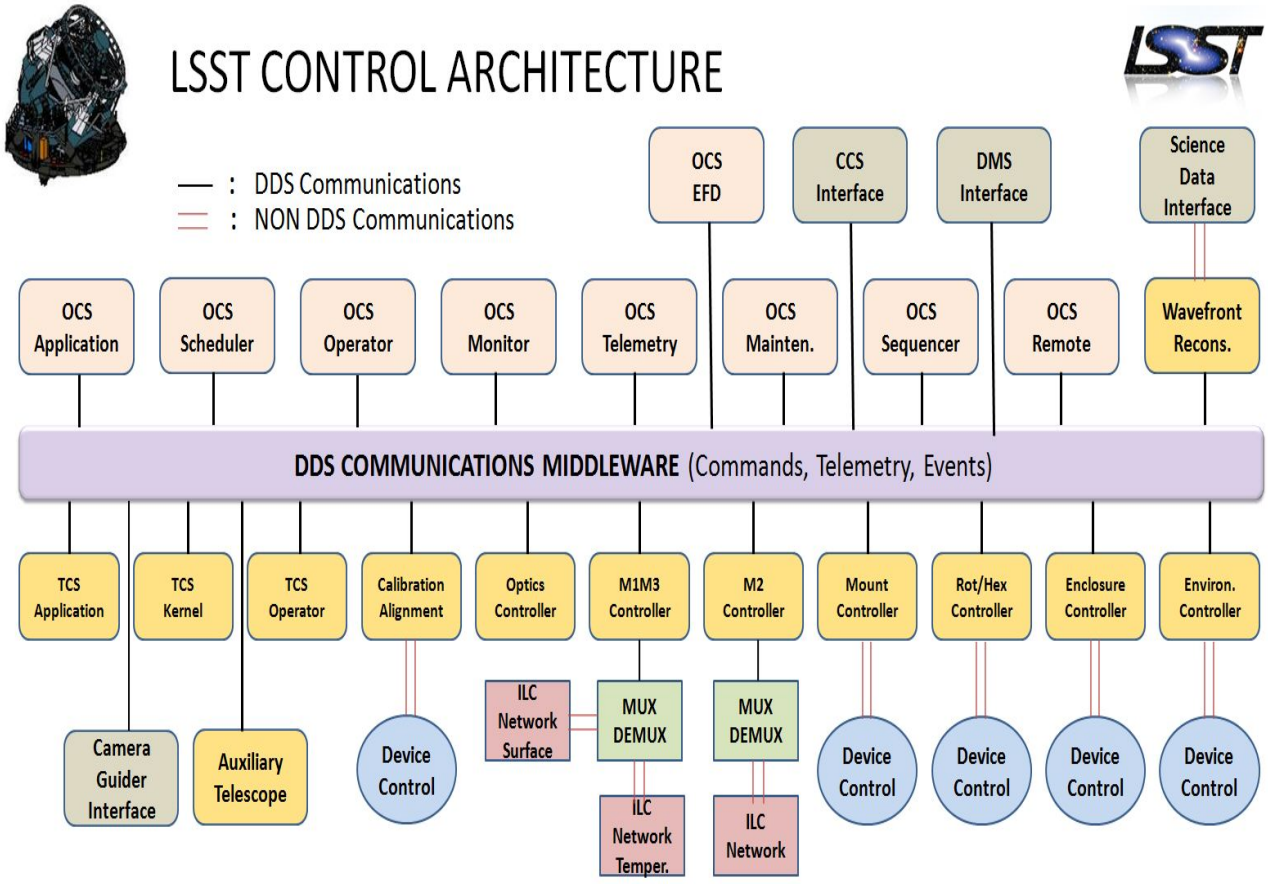

Figure 1. LSST control architecture. The loosely coupled architecture provides for scalability with complex connectivity managed through the data bus. The color coding in this figure has no particular significance.

individually with any device or perform complicated handshake protocols with the principal subsystems of the observatory. The OCS has all the relevant telemetry available through DDS, and by getting the states, values and conditions of the subsystems, it can send the appropriate commands at the right moment to direct the observatory actions according to the observing mode.

The same architecture and control strategy is applied inside OCS between its own components. Each OCS

Figure 2. OCS principal subsystem interfaces. This shows the interface control documents associated with the main control elements of the OCS and subsystems. Not shown are the telemetry and events produced by the OCS itself.

component can subscribe to relevant external and internal telemetry, and publishes new conditions and parameters to realize the OCS activities. This architecture enables a distributed deployment of the OCS software components with a highly decoupled control relationship.

Nominally, subsystems present themselves to the OCS as *commandable entities* and the number of such entities exposed is a contract between the subsystem developers and the OCS software team. All commandable entities present themselves via a well-developed state model.[7] These commandable entities obey the interface control document contract agreed between the developers[8–11] and are subject to rigid change control. Commandable entities obey the generic and business logic command[12] set shown in table 2 plus a set of behavioural commands that are subsystem dependent. Examples of behavioural commands are *ccsSetFilter()* for the *Camera* commandable entity. It is entirely permissible for a commandable entity to have no behavioural commands and just obey the state model (command) triggers. The principal interfaces are shown schematically in figure 2.

In the remainder of this paper, we consider each of the meta-components identified in table 1 in separate sections.

2. APPLICATION

The *Application* meta-component is analogous to the old-fashioned concept of a `main()` routine. It contains the business logic to run the observatory and supports data, science, calibration and engineering observations, operations procedures and smooth transitions between such operating modes. A vital part of the *Application*

Table 2. `Generic` and *business logic* commands of an OCS commandable entity

abort()
`disable()`
`enable()`
`enterControl()`
`exitControl()`
`setValue()`
`standby()`
`start()`
stop()

logic concerns (graceful) error recovery to maximize operations efficiency in the event of (partial or full) subsystem failure. This smart recovery blueprint is currently under development.

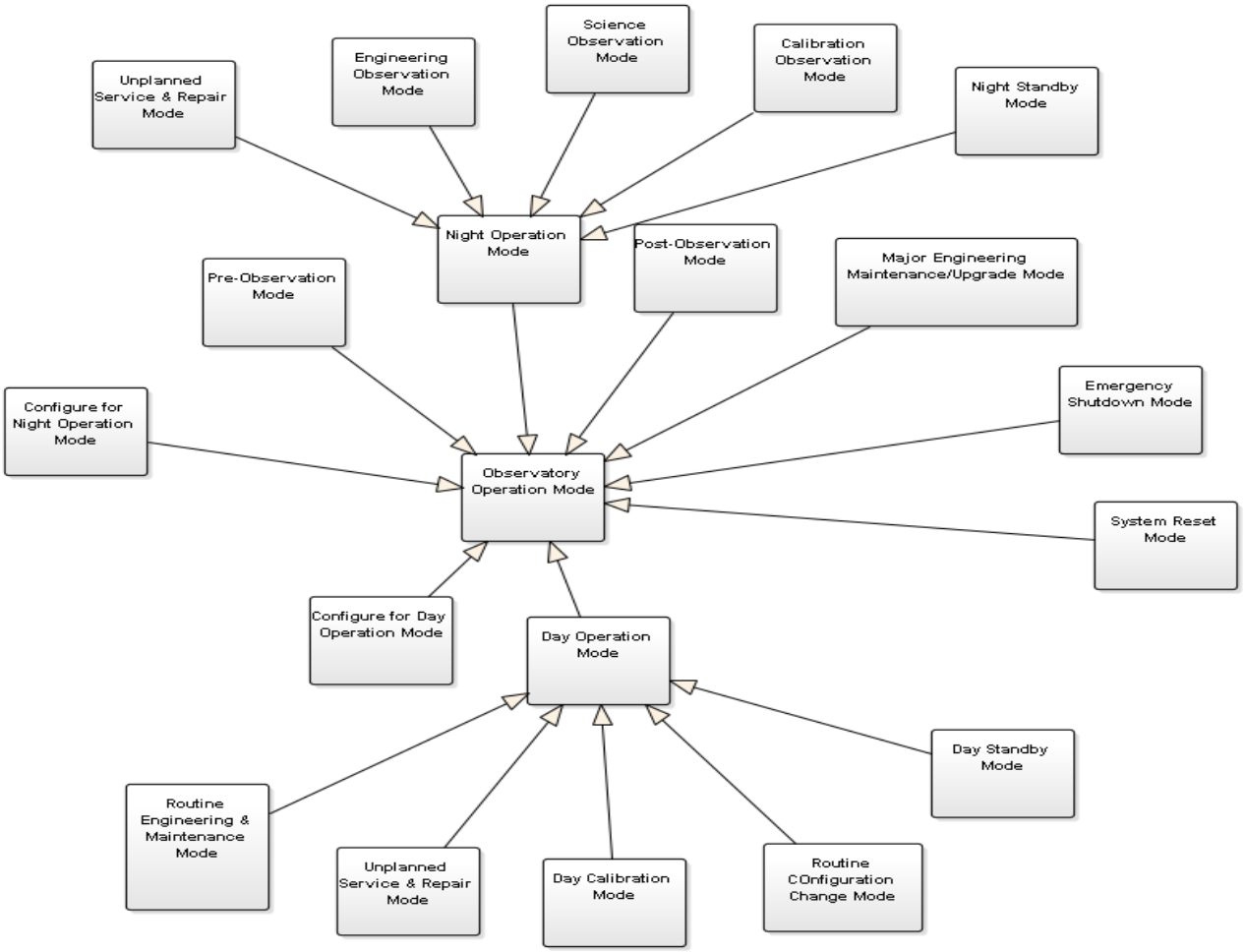

Figure 3. *Enterprise Architect* model of observatory modes. Transitions between these modes are the domain of the *Application* component.

The requirements analysis for this component drives us towards an orthogonal state machine of operations between known states and sub-states. The development of this state machine is ongoing with input provided by the technical operations working group as they have analyzed many aspects of observatory operations[13] and their SysML diagram of observatory modes is shown in figure 3 for reference. As examples, the transition between various operations activities is shown in figure 4 and the actions performed to configure for in-dome calibration are shown in figure 5.

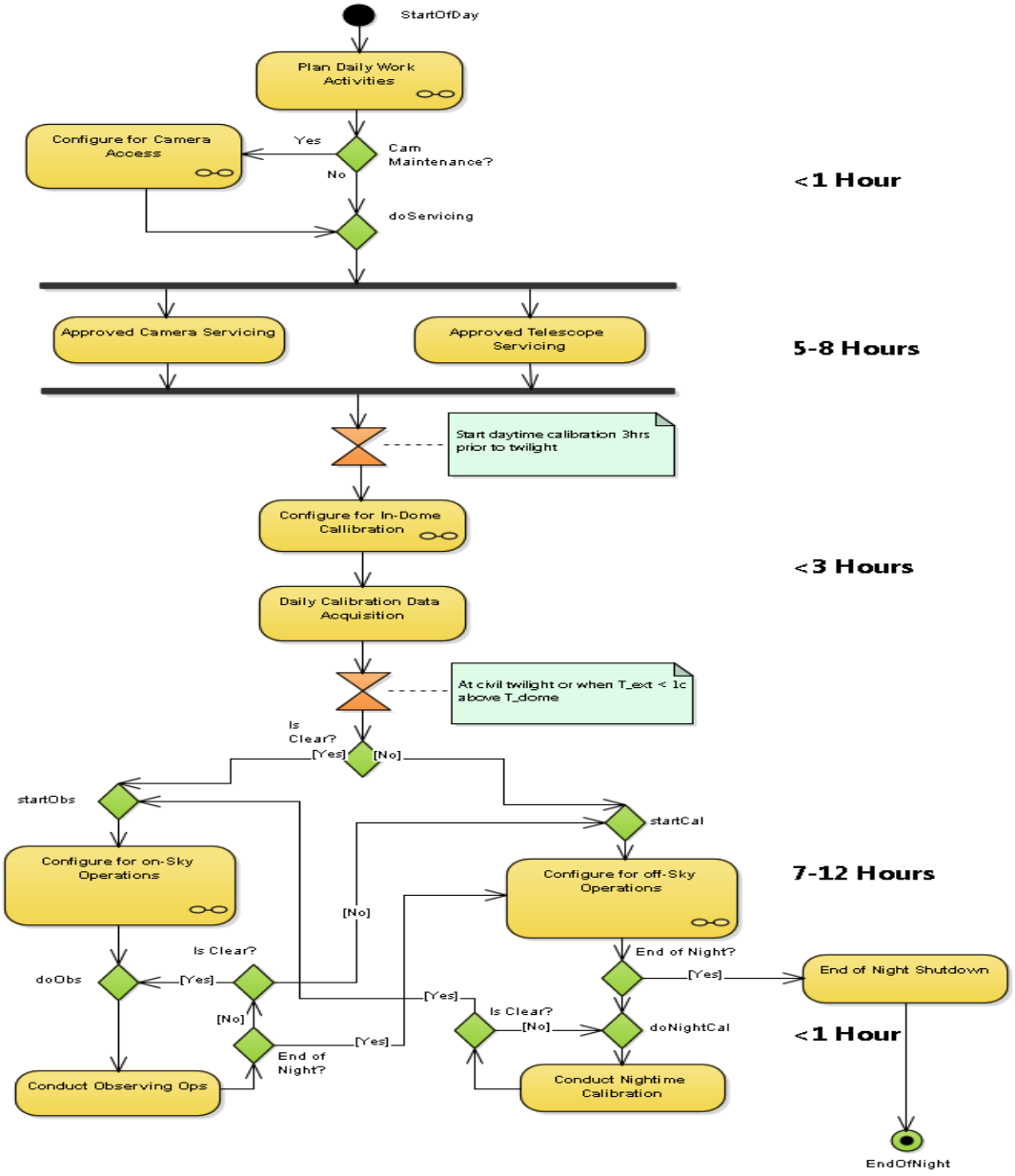

Figure 4. Model of 24-hour summit operations cycle. These are examples of the seamless transitions that the *Application* component will co-ordinate.

A main task of the *Application* meta-component will be to select which observation will be performed and in

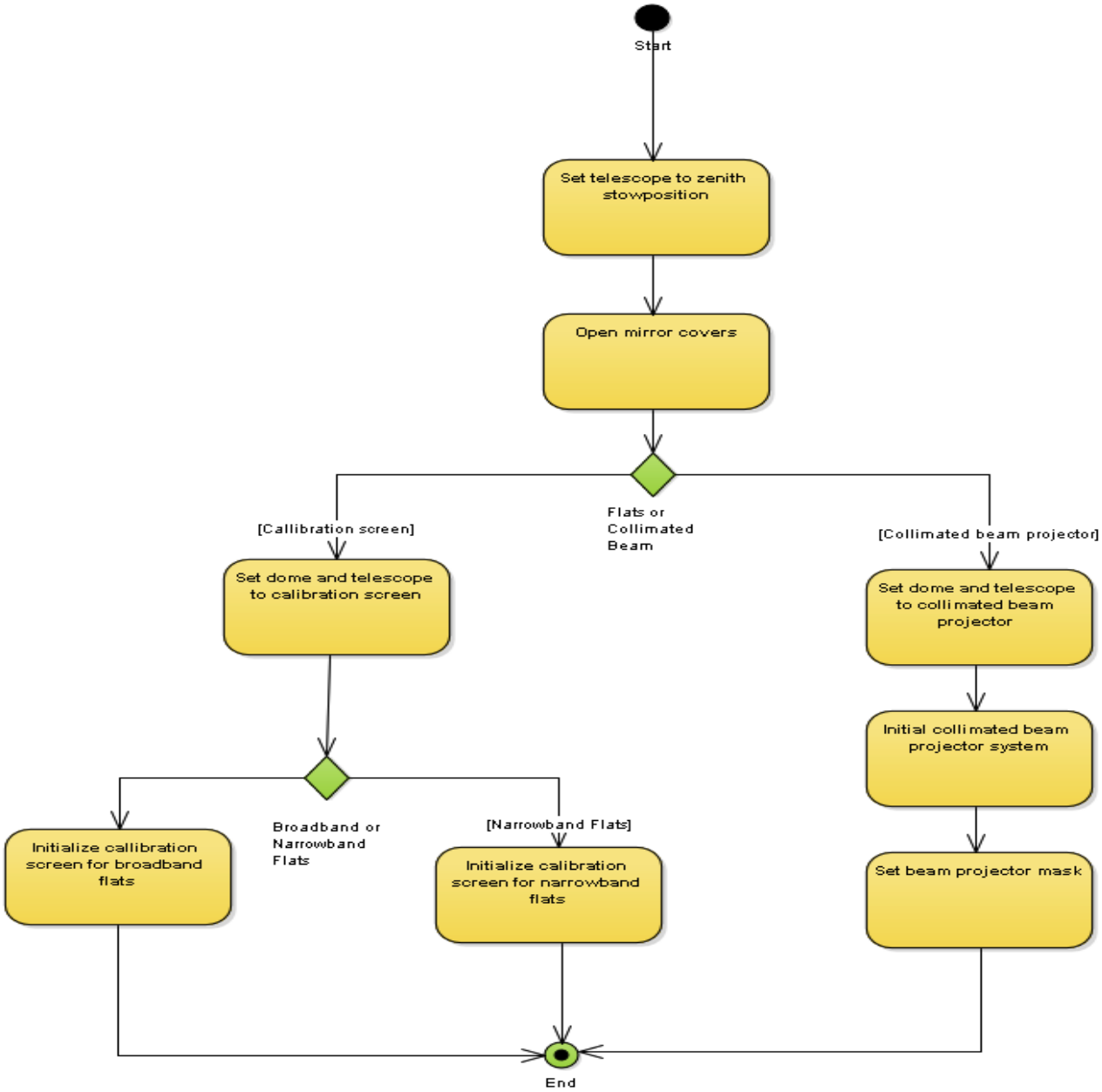

Figure 5. Model of configuring for in-dome calibration. In this diagram, we see specific actions required for different types of in-dome calibration procedures.

what order. Although the *Scheduler* component suggests the next visit, the *Application* is under no obligation to select this choice. Circumstances where it will override the *Scheduler* option will be rare but can encompass targets of opportunity and *ad hoc* calibration or engineering sequences (especially during commissioning).

When an observation is available, the *Application* will be notified by a well-known event. The event payload will include the location of an XML configuration specifying the details of the required observation. From this payload, and a scientist-supplied template file for the requested observation type, the *Application* script builder will be able to create, dynamically, a complete script to perform the observation tagged with a unique observation

identifier. This is similar to the NEWFIRM[14] approach but with the business logic being contained with the *Application* meta-component (rather than the end user GUI interface).

This unique observation identifier can be used as an index for observation management. This can be done with a modest piece of code*:

```
>> from astropy.time import Time
>> def getObservationIdentifier():
>>   return float(''.join("{at:.17f}".format(at=Time(str(Time.now().iso)).mjd)))
>> getObservationIdentifier()
57496.91866954861325212
```

This identifier can then be used as a dictionary key with the value equating to the associated script (held within, for example, `<script-path>/57496.91866954861325212.scr`). Since the keys appear as a collection of floating-point monotonically increasing numbers, a FIFO can be created by asking for the minimum-value key (from the set) and a LIFO by asking for the maximum-value key (no matter what order they are stored in). For OCS purposes, a FIFO is optimal and we can introduce any urgent observation (one that has to be done before anything else in the FIFO) by unary negation of the observation identifier.

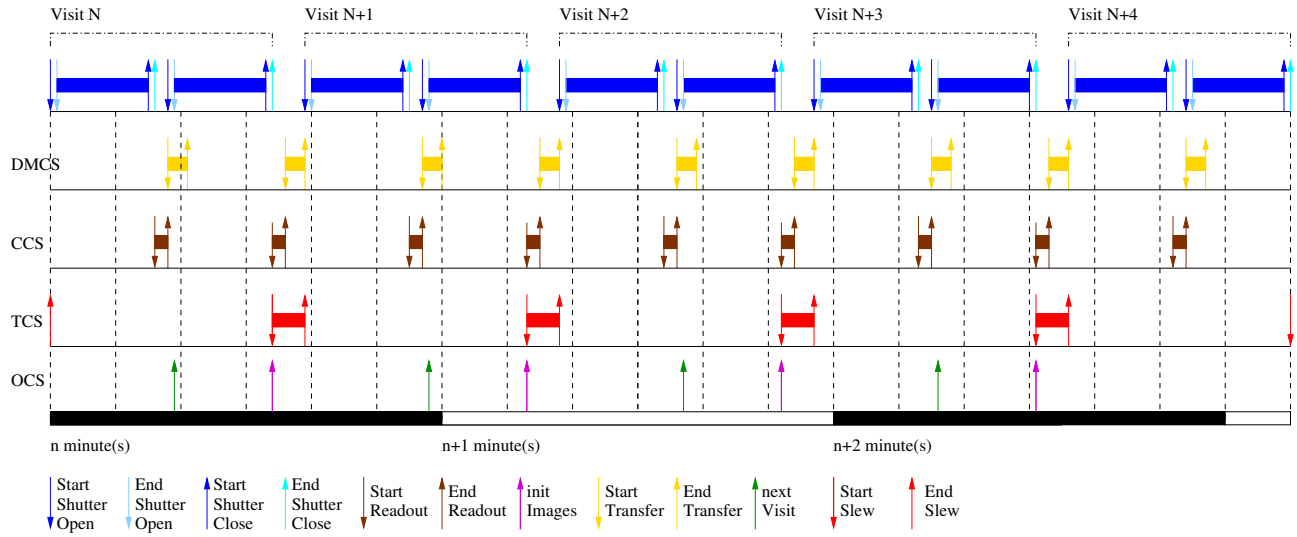

Figure 6. Median timing diagram for advance notice of pointings in the steady-state survey. This to-temporal scale diagram shows the timing required for signals to conform to the DMCS advance-pointing requirement. This is the median case but we have diagrams for the mean and fastest cases, too.

The design and development of the management of this queue, particularly with respect to the requirement on the 'advance notice of (telescope) pointings' for the data management control system, is being guided by timing diagrams like that shown in figure 6.

3. ENGINEERING FACILITY DATABASE

The engineering and facility database (EFD) design is predicated on the project requirement to maintain a complete and comprehensive archive of all system information (operational, environmental and configurational) in a coherent and easy to access form.[6] The database will be used in a near real-time access mode, as a source for daily system status reporting as well as long-term historical trending purposes.

*A more robust piece of code will be developed that ensures the uniqueness of this value. A similar mechanism has been in place since 1994 on many other instruments on many telescopes developed by the principal author and has never been known to produce a duplicate number.

The database will record all telemetry (raw and derived) issued by all the subsystems of LSST. It will also record the complete command and configuration history of each subsystem.

An EFD cluster is comprised of a set of commercial-off-the-shelf (COTS) rack mount servers, each supporting 4×8 Tb storage units currently baselined as SATA HD, but solid state may be substituted depending upon price/performance ratios. Two identical instances of the EFD cluster are operational at all times. One resides on the summit, and the other at the base facility. The EFD data will also be replicated at the archive center(s).

The EFD data is comprised of the following types of objects:

- Telemetry datastream records;
- Command and response records;
- Alert and log records;
- Configuration records;
- Derived data records;
- Binary large-file objects (images, PDFs, spreadsheets *etc*).

Each type of object will typically be produced in many flavours by each LSST subsystem. In order to achieve a coherent system and database design, there is a one-to-one correspondence between the atomic items in each telemetry datastream (mediated by the middleware), and the telemetry datastream records in the EFD. Additional higher level information (derived data) may also be stored.

Large objects are not stored natively in the database. Although modern SQL databases have this capability, it is generally very inefficient in both storage space and latency. The EFD will instead store large-file objects as independent files with the exact type determined by data itself *e.g.,* `.fits`, `.mpg` *etc*. The data access layer provided by the SAL automatically integrates a record of the details of each large object file, as it is mandatory for the creating subsystem to also publish an event describing the object.

A subsequent event is published by the EFD cluster once the object has been sucessfully copied and validated (checksum, update date, version *etc*). Users of large object data items always retrieve them from the EFD using a uniform resource locator (URL) as specified on the EFD announcement event, typically retrieving the entire object at once. This URL conforms with RFC-2396. The supported set of file transfer protocols is a subset of those supported by the `cURL` library (at a minimum to include `http`, `https`, `scp`, `file` and `smb`).

Several classes of queries are supported:

Recent history (operator display functionality) :
: Many instances of this query class will be fulfilled by the middleware without needing to refer to the database. However, many queries requiring larger datasets (*e.g.,* last hour), will normally be fulfilled by the on-site EFD;

24-hour period (constructing daily reports) :
: This query class would normally be processed on the base EFD cluster, but can also be processed on-site if required;

Large (days, weeks, months) :
: This query type will normally be expected to be processed at the data center where large resources are located;

Real-time (last recorded value) :
: This query type will normally be fulfilled by the middleware without needing to refer to the database. The possibility of forcing a query is expected to be a rare special case.

All records in the EFD are time tagged with 2 items: the local machine time when the corresponding information was generated and the arrival time on the EFD cluster. The fidelity of the system time is maintained cluster wide using precision time protocol (PTP) to within 1 ms.

4. MAINTENANCE

The *Maintenance* meta-component derives its functionality from the following observatory system specification (OSS[15]) requirement:

> '*The LSST system shall be designed for maintainability of components, and a maintenance plan shall be implemented to achieve the required survey performance during the life span.*'

This OCS component is the software that supports the maintenance activities of the observatory, collecting and storing the relevant telemetry and keeping track of routine maintenance operations. It provides functionality and interfaces for generating reports, logs, and supporting the management of the maintenance plans. This support is utilized by a plan for the facilities and maintenance effort, with the goal of long term sustained operations as the primary objective.

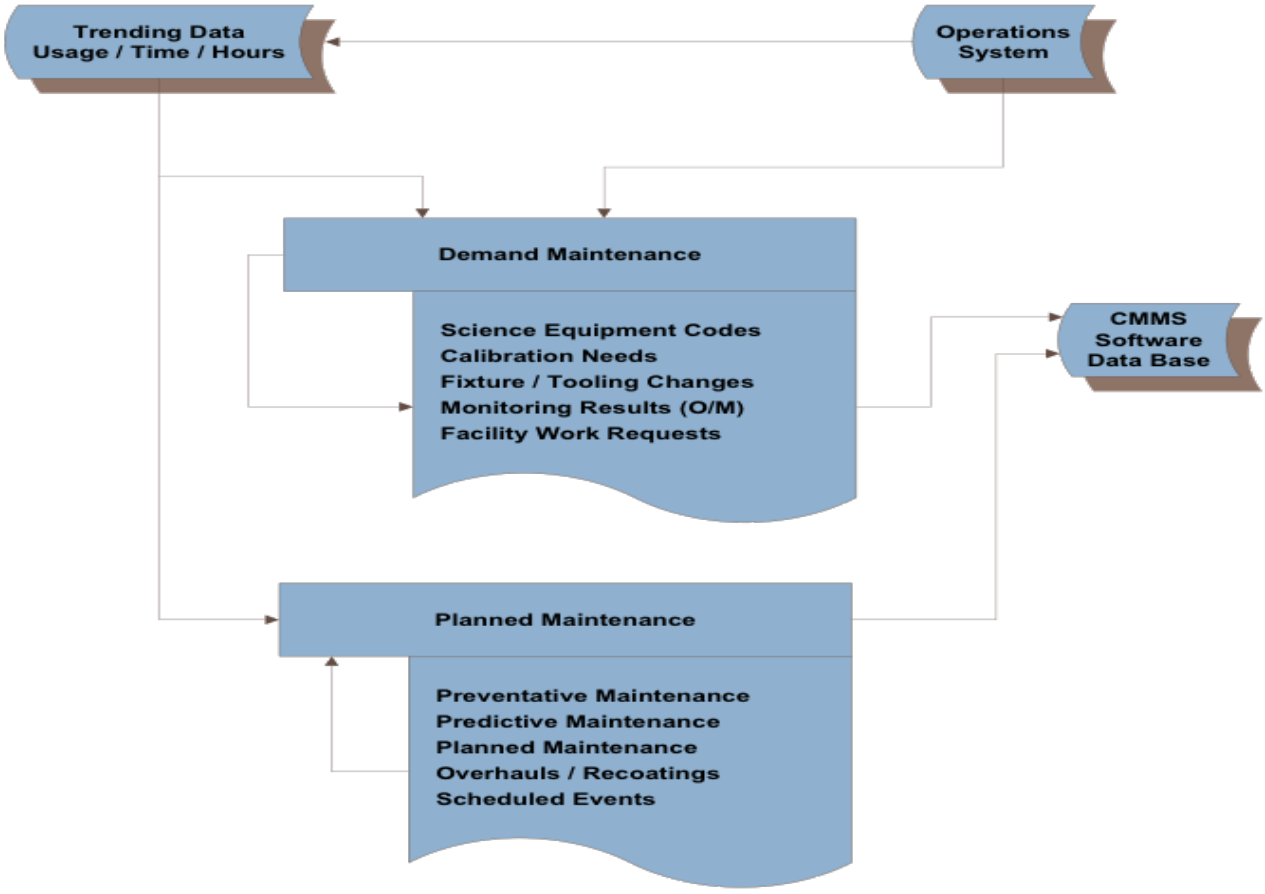

Figure 7. LSST CMMS database implementation and interface with operations system. The CMMS is linked to the OCS middleware to enable access to the usage logs of the operational systems.

The heart of the plan is a commercial computerized maintenance management system (CMMS[†]), now in the process of being selected. A CMMS software package maintains a computer database of information about an organization's maintenance operations. This CMMS packages provide capabilities to generate preventive and predictive maintenance activities, schedule calibration and tests, measure downtime, track tooling and parts, analyze trending data on usage and failures, among other features.

[†] https://www.wbdg.org/om/cmms.php

The implementation of the *Maintenance* component consists, then, of a CMMS package and interface software linked to the OCS communications middleware, for interactions with the operations system. This is illustrated in figure 7 showing this integration of the operations system with the CMMS functionality.

5. MONITOR

The *Monitor* meta-component provides the tools and visualization of telemetry and events to allow operators to see the current state of the observatory and trends emerging from the data. In every sense, the *Monitor* meta-component can be thought of as a *data science* component. It is not concerned with visualization of the pixel data for that is handled by a different component, being developed by the camera and data management subsystems. Nor is it concerned with the collection of fundamental data as all telemetry and events that pass through the SAL are automatically logged into the engineering and facility database. Aggregated data (from various sources) that contribute to the overall status of the observatory (and the OCS) will, typically, be created by the *Telemetry* meta-component.

Therefore, we can breakdown the *Monitor* into the following components:

1. Alarm production display and management.
   It is vital that alarms are clearly visible to the 'operators' of the telescope (some of whom will be on-site and some remote). This component will utilize code from many aspects of the operator-remote development as we would wish this component to be scalable from desktop through laptops to smart phones. We are recruiting in this area but it is equally possible that we will contract with outside agencies to deliver some of this functionality, *e.g.,* INRIA in Chile, and we have already opened a dialogue with such companies. We are also looking at other developments such as those on `www.simile-widgets.org`.

2. Report generation.
   Reports can be generated automatically (at the end of the night, say) to facilitate day-to-day operations and engineering planning for the observatory or may be bespoke for specific engineering or science activity. This component is relatively uncomplicated and is based upon scientists and engineers identifying which database queries are relevant to the report in hand. The report output should be made available as both HTML and PDF. This will, in all likelihood, be an activity for the commissioning phase of the project.

3. Trend analysis.
   This is the most interesting aspect of the *Monitor* and we will utilize a *complex event processing* (CEP[16]) paradigm which transforms a given set of (usually diverse) inputs into a known output stream[‡] *i.e. viz.,* from source(s) to effect. We anticipate that this particular development will occur later in the construction cycle as we evaluate suitable software packages and gain expertize with observatory operations during the early commissioning period.

6. OPERATOR-REMOTE

To present a consistent view of the system to users the, formerly separate, *Operator* and *Remote* components have been combined into a single, unified meta-component. Since we will have both local and remote users, we will implement the interface via a set of web pages with appropriate authentication and security protocols. The back-end software of this development is inchoate but we did evaluate several programs from other observatories regarding many aspects of their software and UI/UX paradigms[17,18] and hope to benefit from such open source collaborations.

However, we can say that the inputs that define the web pages will be based upon a well-defined XML schema (`ocs.xsd`[§]). As proof-of-concept for this functionality, we utilized a tool called *XSDForms*, which ingests a schema and automatically produces a web page based upon it, to present a *ccsTakeImages()* interface on a web page as shown in figure 8. Pressing the <SEND> button on this web-GUI, produces the following output:

[‡]An example of such a paradigm, in current practice, is credit-card expenditure tracking which leads to fraud alerts when the purchase appears beyond the bounds of normal activity.

[§]Available at https://github.com/lsst-ts/ocs_xml.git

```
<ccsCommand Subsystem='Camera' WaitForCompletion='True' RecordID=1>
  <NumberOfImages>2</NumberOfImages>
  <ExposureTime>15.0</ExposureTime>
  <ShutterCondition>True</ShutterCondition>
  <ScienceReadout>True</ScienceReadout>
  <GuiderReadout>False</GuiderReadout>
  <WfsReadout>False</WfsReadout>
  <ImageSequenceName>TestProofOfConcept</ImageSequenceName>
</ccsCommand>
```

When this file is created on a back-end server, a SAL event informing the other components of the system that a new observation is available will be triggered.

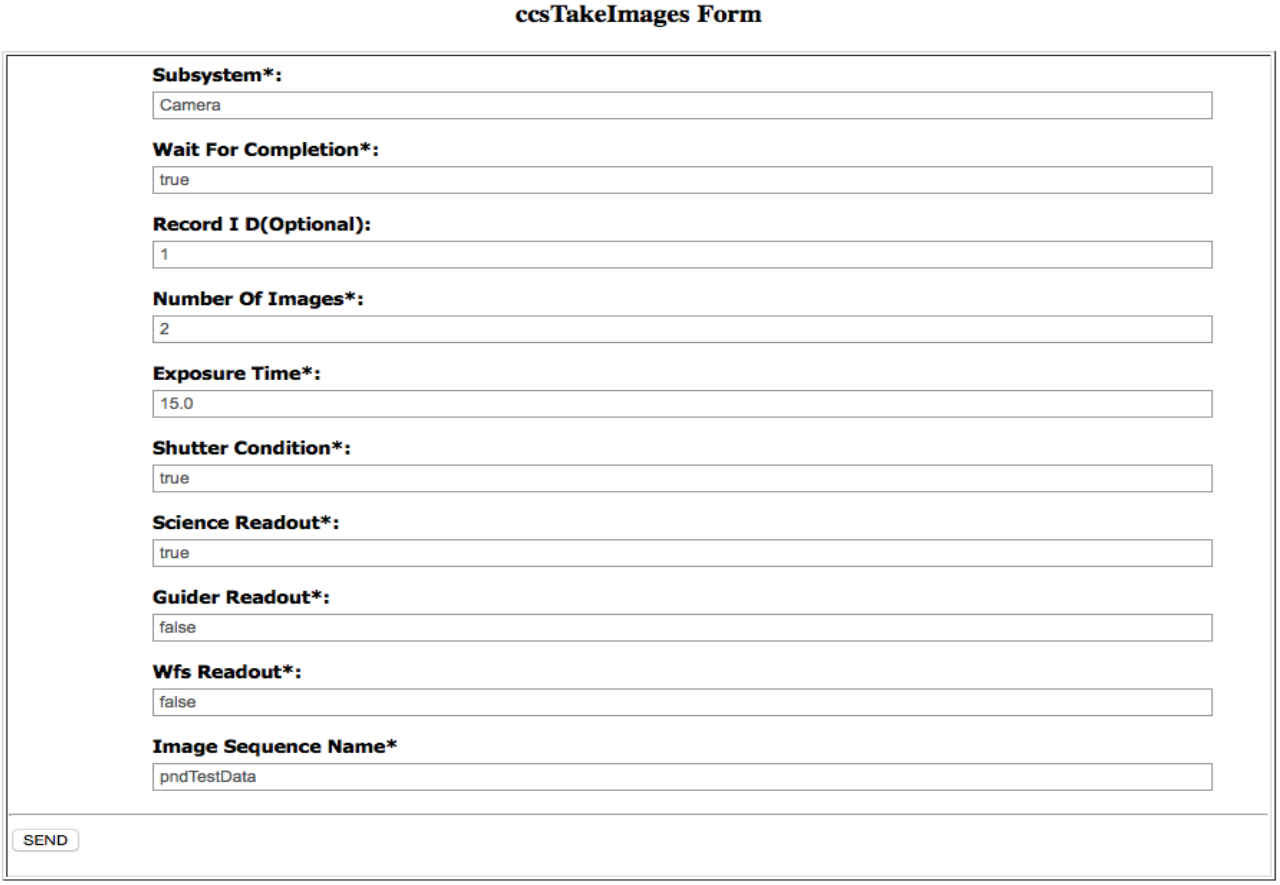

Figure 8. *ccsTakeImages* Command via an *XSDForms* interface. This web page is created automatically by the *XSDForms* software simply by ingesting the associated, well-formed, schema.

Even though we will strive to achieve the same 'look and feel' for the OCS interfaces, we accept that—with so many subsystems and diverse vendors supplying the software interfaces—engineering interfaces will look different to the preferred OCS visual designs. Such interfaces will be hidden from regular and remote users unless authenticated as system experts.

7. SCHEDULER

The *Scheduler*[19] is the OCS meta-component that produces the targets for the observatory, implementing the survey. Due to the variety of science goals and the operational constraints of the LSST observatory, the *Scheduler* was envisioned as a fully automatic and dynamic component that—according to the current conditions of the observatory and the environment, the past history of observations and the multiple goals from the multiple science programs defined in the survey—determines the next target at real-time.

The need for such a *Scheduler* for the scientific success of the LSST was identified early on the project. In addition, the scale of the survey and the fact that the mission is about multiple sciences missions simultaneously, made the *Scheduler* a high-risk component. In order to mitigate this risk, prototypes of the *Scheduler* were developed from the beginning of the research and development phase in the project, working in a highly detailed simulation environment. The result is a detailed design for the *Scheduler*, and a construction plan that incorporates the simulation aspect for constant validation.

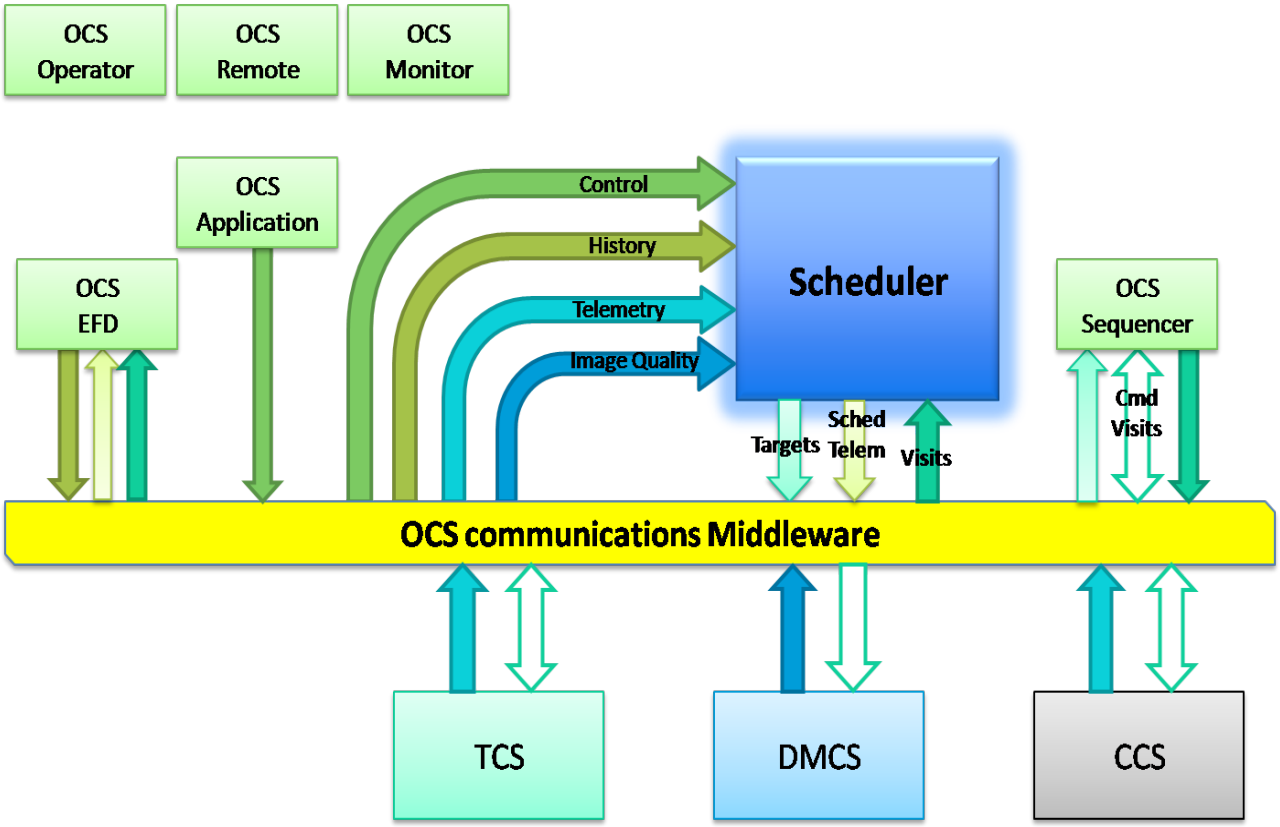

Figure 9. *Scheduler* component in the context of the OCS. Here we see that the component ingests telemetry and data from a variety of subsystem using the DDS bus and outputs the most opportune target for the next visit.

Figure 9 illustrates the context of the *Scheduler* as part of the OCS. The *Application* controls the operational modes, the EFD provides the history of previous observations during start-up, TCS and CCS provide the telemetry for the current observatory conditions. The environment monitoring control system (EMCS) provides telemetry about the environmental conditions, and the DMCS provides the feedback with measured image quality parameters from past observations. All this information is computed by the scheduling algorithms to produce the targets, main output from the *Scheduler*. The *Sequencer* executes those targets, and the actual observations are then notified back to the *Scheduler* the register the visits and adjust the priorities for the upcoming targets. *Scheduler* telemetry is also produced, with information regarding the decisions processes for monitoring the algorithms.

8. SEQUENCER

The *Sequencer* is the software meta-component that orchestrates the transmission of the commands to the observatory subsystems in order to follow the series of targets provided by the scheduler, in the fully automatic survey mode, as well as in scripted observations, calibration or engineering. We have reduced this to a set of one-on-one components for each commandable entity and proved the concept using Python classes.

In short, we have developed a class (*ocsGenericEntity*) that provides the base functionality of the state model to all commandable entities. We have further developed a camera class (*ocsCameraEntity*) that inherits from this base class and adds the behavioural commands associated with the *Camera* commandable entity. Extending this base class to other commandable entities with behavioural commands is transpicuous but will circumscribe the timeline for delivery.

Indeed, so successful is this model that, in a very real sense, the *Python virtual machine can be considered the sequencer itself* with Python providing the scripting capabilities required by the users. For example, the *ccsTakeImages()* XML output from the web-GUI of section 6 can be translated into the following script based upon a template:

```
# import the ocsCameraEntity class
from ocsCameraEntity import *
# create a camera instance
camera = ocsCameraEntity('CCS','Camera')
# turn off simulation which is enabled by default
camera.simulation = False
# issue command ccsTakeImages(numImages,ExpTime,Shutter,Science,Guider,WFS,name)
camera.ccsTakeImages(2,15.0,'Open',True,False,False,'TestProofOfConcept')
```

The use of `argparse` can provide each of the developed classes with an instant command line interface satisfying that OCS requirement.

8.1 Operator-Remote Re-Visited

We mentioned in section 6 the development of a consistent set of web interfaces to operator and users. Until such consistent user interfaces are developed, we can re-use the classes mentioned above wrapped with a `tkinter` GUI to create an operator console for testing and commissioning purposes. This is shown in figure 10 where several generic commandable entities are shown and can be toggled on/off using the appropriate checkbutton on the left-hand side of the interface.

9. SERVICE ABSTRACTION LAYER

The LSST middleware is based on a set of software abstractions which provide standard interfaces for common communications services. This is driven by the observatory requirement for communication between diverse subsystems, implemented by different contractors, and comprehensive archiving of subsystem status data.

The service abstraction layer (SAL[6]) is implemented using open source packages that implement open standards of the data distribution service (DDS) for data communication and standard query language (SQL) for database access. Specifically, the (PrismTech) *OpenSplice* community edition of DDS provides a LGPL distribution which may be freely re-distributed. The availability of the full source code provides assurances that the project will be able to maintain it over the full 10 year survey, independent of the fortunes of the original providers.

For each subsystem, abstractions for each of the telemetry data streams, along with command/response and events, have been agreed with the appropriate component vendor (*e.g.,* Dome, TMA, Hexapod) and captured in interface control documents (ICDs). The definition of instances of these abstractions is tightly controlled by reference to a system dictionary. All code referencing them is automatically generated and includes real-time consistency checking on a per-transaction basis. All command transactions, telemetry and event messages are automatically stored in a system wide facility database system.

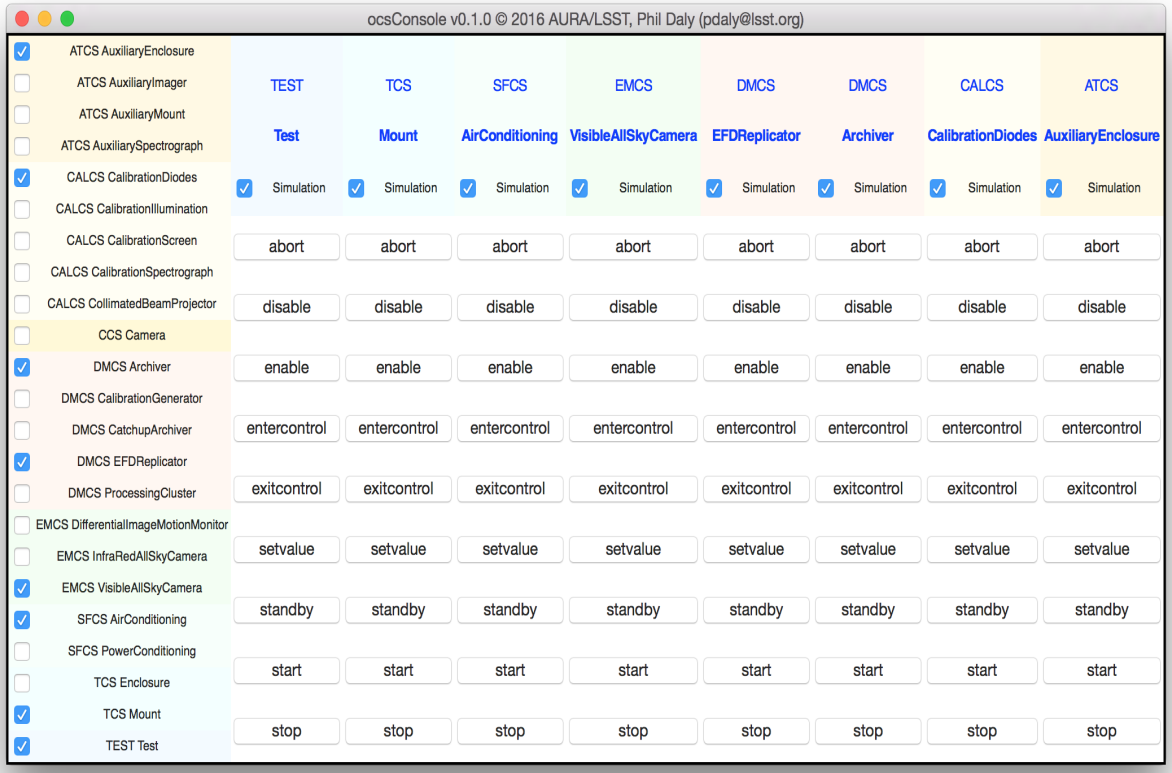

Figure 10. A tkinter console re-using sequencer classes. This is a GUI wrapper to the classes developed for the *Sequencer* and provides a dynamic OCS console.

The full set of defined transactions (per subsystem) is formally described using an XML Schema. The XML is then used to automatically generate DDS compliant interface definition language (IDL) code. The IDL is also extended with a set of SAL specific metadata which provides real-time consistency checking, clock slew detection, and full traceability on a per message basis.

The code base to support the complete set of SAL mediated communications—languages include C++, Java, Python and LabVIEW—is auto-generated, along with a comprehensive set of unit tests and extensive documentation. Application writers need only the appropriate runtime packages (SAL, DDS shared libraries or `jar` archives) in order to communicate with any other subsystem. Another paper[6] provides intimate details of this software infrastructure layer.

10. TELEMETRY

Telemetry is the meta-component that performs utility functions on the telemetry stream that are of common interest to the OCS components (*e.g.,* figures of merit *etc*). This will be developed *ad hoc* as the need for such metadata arises.

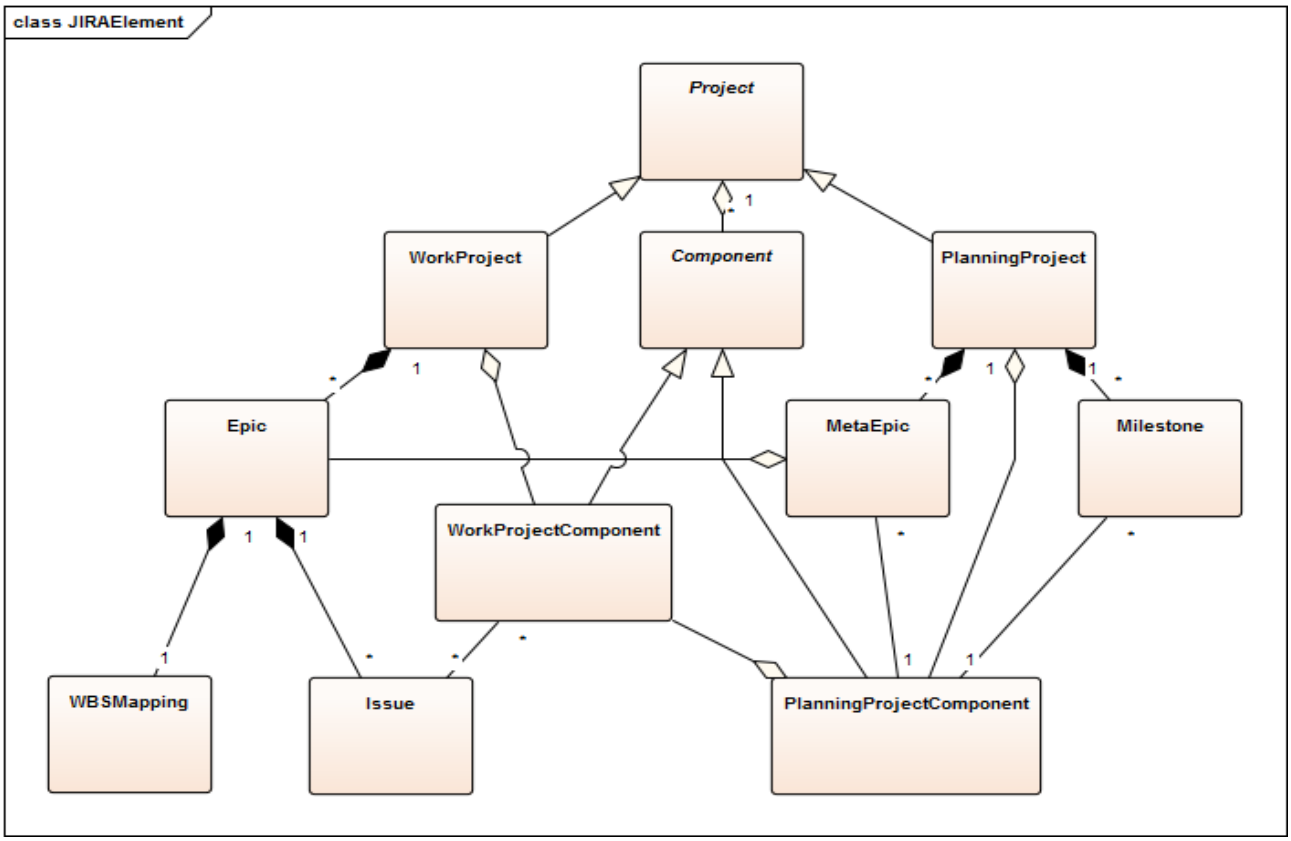

Figure 11. Relationship between JIRA and the project (in effect, PMCS) for telescope and site software planning. The planning project contains a set of meta-epics, which have a composition relationship. The epics themselves have a aggregation relationship to the meta-epics.

11. TIMELINE AND PLANNING

All elements of the project are subject to critical deadlines as outlined in the overall project plan[¶]. For telescope and site software development, we have adopted several tools for tracking and delivery of meta-components within an earned value paradigm. A complementary approach is described in another paper.[20]

We are using project-wide tools such as *Atlassian* `Confluence` and `JIRA` with code builds utilizing both a private `Stash` repository (for proprietary code) and `GitHub` (for open source code). We have recently added a `Jenkins` continuous integration server[‖] to automate this delivery. A new hire is also tasked with software testing.

To provide input into the global project management control system (PMCS) and track issues within, we have set up a 'Telescope and Site Software Planning Project' (TSSPP) and a 'Telescope and Site Software' (TSS) JIRA-based tracking system. We show a model of this in figure 11. Meta-epics within the TSSPP scope are reduced to a series of epics within TSS and monthly sprints are held on these items.

12. CONCLUSION

In this paper we have discussed all meta-components within the OCS domain and their current status. We have outlined our plans for these meta-components and have schedule and budget for timely delivery to the project in the construction, integration and early and full commissioning phases.

[¶] http://lsst.org/about/timeline
[‖] http://ts-ci.lsst.org

REFERENCES

[1] Kahn, S., "Final Design of the Large Synoptic Survey Telescope," in [*Ground-based and Airborne Telescopes VI*], Hall, H. J., Gilmozzi, R., and Marshall, H. K., eds., *Proc. SPIE* **9906**, in press (2016).

[2] Kurita, N. et al., "Large Synoptic Survey Telescope camera design and construction," in [*Advances in Optical and Mechanical Technologies for Telescopes and Instrumentation*], Navarro, R. and Burge, J. H., eds., *Proc. SPIE* **9912**, in press (2016).

[3] Juric, M. et al., "The LSST Data Management System," in [*Astronomical Data Analysis Software & Systems XXV*], Lorente, N. P. F. and Shortridge, K., eds., *ASP Conf. Series*, arXiv:1512.07914 in press (2016).

[4] Schumacher, G. and Delgado, F., "Observatory Control System Requirements," *LSST Systems Engineering Collection* **62** (2016). http://ls.st/LSE-62.

[5] Schumacher, G. and Delgado, F., "The Large Synoptic Survey Telescope OCS and TCS models," in [*Modeling, Systems Engineering and Project Management for Astronomy IV*], Angeli, G. Z. and Dierickx, P., eds., *Proc. SPIE* **7738**, 77381E (2010).

[6] Mills, D., "LSST communications middleware implementation," in [*Ground-based and Airborne Telescopes VI*], Hall, H. J., Gilmozzi, R., and Marshall, H. K., eds., *Proc. SPIE* **9906**, in press (2016).

[7] Lotz, P. J. et al., "LSST control software component design," in [*Software and Cyberinfrastructure for Astronomy*], Chiozzi, G. and Guzman, J. C., eds., *Proc. SPIE* **9913**, in press (2016).

[8] Johnson, T. et al., "OCS-Camera Software Communication Interface," *LSST Systems Engineering Collection* **71** (2015). http://ls.st/LSE-71.

[9] Dubois-Felsman, G. et al., "Data Management – OCS Software Communication Interface," *LSST Systems Engineering Collection* **72** (2014). http://ls.st/LSE-72.

[10] Schumacher, G., "OCS Command Dictionary for the Telescope," *LSST Systems Engineering Collection* **73** (2011). http://ls.st/LSE-73.

[11] Ingraham, P. and Daly, P. N., "Calibration Hardware and Environmental Monitoring Subsystem Control Iterface," *LSST Systems Engineering Collection* **139** (2016). http://ls.st/LSE-139.

[12] Daly, P. N., "OCS Commanding and Scripting," *LSST Telescope & Site Collection* **237** (2015). http://ls.st/LTS-237.

[13] Selvy, B. M. and Claver, C., "Using model based systems engineering for the development of the Large Synoptic Survey Telescope (LSST) concept of operations," in [*Modeling, Systems Engineering and Project Management for Astronomy VI*], Angeli, G. Z. and Dierickx, P., eds., *Proc. SPIE* **9911**, in press (2016).

[14] Daly, P. N. et al., "The NEWFIRM Observing Software: From Design To Implementation," in [*Advanced Software and Control for Astronomy II*], Bridger, A. and Radziwill, N. C., eds., *Proc. SPIE* **7019**, 701913 (2008).

[15] Claver, C. F. et al., "Observatory System Specifications," *LSST Systems Engineering Collection* **30** (2016). http://ls.st/LSE-30.

[16] Luckham, D., [*The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*], Addison-Wesley Professional (2002). ISBN-10 0201727897.

[17] Eiting, J. et al., "Graphical User Interfaces of the Dark Energy Survey," in [*Software and Cyberinfrastructure for Astronomy*], Radziwill, N. C. and Bridger, A., eds., *Proc. SPIE* **7740**, 774012 (2010).

[18] Pietriga, E. et al., "A Web-based Dashboard for the High-level Monitoring of ALMA," in [*Software and Cyberinfrastructure for Astronomy III*], Chiozzi, G. and Radziwill, N. C., eds., *Proc. SPIE* **9152**, 91521B (2014).

[19] Delgado, F., "The LSST scheduler from design to construction," in [*Observatory Operations: Strategies, Processes and Systems VI*], Peck, A. B., Seaman, R. L., and Benn, C. R., eds., *Proc. SPIE* **9910**, in press (2016).

[20] Kantor, J. et al., "Agile software development in an earned value world: a survival guide," in [*Modeling, Systems Engineering and Project Management for Astronomy VI*], Angeli, G. Z. and Dierickx, P., eds., *Proc. SPIE* **9911**, in press (2016).