LSST control software component design

Paul J. Lotz[a], Gregory P. Dubois-Felsmann[b], Kian-Tat Lim[c], Tony Johnson[c], Srinivasan Chandrasekharan[a], David Mills[a], Philip Daly[a], Germán Schumacher[d], Francisco Delgado[a], Steve Pietrowicz[e], Brian Selvy[a], Jacques Sebag[a], Stuart Marshall[c], Harini Sundararaman[a], Christopher Contaxis[a], Robert Bovill[a], Tim Jenness[a]

[a]Large Synoptic Survey Telescope, 950 N Cherry Ave, Tucson, AZ, USA 85743;
[b]Infrared Processing and Analysis Center, California Institute of Technology, Pasadena, CA, USA;
[c]SLAC National Laboratory, Menlo Park, CA, USA;
[d]Large Synoptic Survey Telescope, La Serena, Chile; [e]National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, IL, USA

ABSTRACT

Construction of the Large Synoptic Survey Telescope system involves several different organizations, a situation that poses many challenges at the time of the software integration of the components. To ensure commonality for the purposes of usability, maintainability, and robustness, the LSST software teams have agreed to the following for system software components: a summary state machine, a manner of managing settings, a flexible solution to specify controller-controllee relationships reliably as needed, and a paradigm for responding to and communicating alarms. This paper describes these agreed solutions and the factors that motivated these.

Keywords: LSST, control software, component, state machine, settings, alarms, control request handling

1. INTRODUCTION

We describe commonalities of design adopted for software components in the Large Synoptic Survey Telescope[1] (LSST) system. These commonalities we view as essential to ensure all components work effectively together to achieve the system goals. In particular, we seek to facilitate communication between teams during design and between software components in the system. In addition, we focus on areas that represent the best return on investment in terms of robust software components. These common areas include elements of component design, most notably shared summary states and extension methodologies, a related analysis of engineering behaviors, a common approach to handling settings, a means for control request handling, and guidelines for handling alarms. All these segments except for control request handling build on earlier work.[2-6]

2. COMPONENT DESIGN

LSST software design embraces the system concept that "each system consists of a set of components that work together to achieve a larger goal."[7] To minimize risk by best ensuring that the components will work together, LSST has adopted a component design that provides core agreements on interface definitions and common behaviors. LSST anticipates, based on team experience on previous projects, that a common design will facilitate interface discussions, reduce debug time, and result in a robust, correctly-working system. Additionally, users will have a similar experience when working with components that share these commonalities when working with LSST components throughout the system.

This paper presents common component design related to state machines, support for engineering, settings, control request handling, and alarm handling.

3. STATE MACHINE DESIGN

Since the reactive systems (which respond to events, vs. transformational systems) that together form the larger system exhibit state-based behaviors, we describe these with state machines. A state machine is "a model of computation

Software and Cyberinfrastructure for Astronomy IV, edited by Gianluca Chiozzi, Juan C. Guzman, Proc. of SPIE Vol. 9913, 991309 · © 2016 SPIE
CCC code: 0277-786X/16/$18 · doi: 10.1117/12.2231796

consisting of a (possibly infinite) set of states, a set of start states, an input alphabet, and a transition function which maps input symbols and current states to a next state. Usually understood to be a finite state machine."[8]

In this context, we are concerned with behavioral state machines (vs. protocol state machines).[9]

A state is "a system situation in which an invariant is valid."[10] We use state machine models to describe how systems behave in response to triggers, and in particular how these behaviors vary according to state.

A component in the LSST system defines its interface in terms of external triggers and the consequent effects of these. LSST requires a state machine description as part of the component interface definition. While executing, each component publishes its summary state and detailed state. With this information a client (a View or another software component) can know which triggers are permissible in the present context and how the component will respond to these, and can display menus and interact accordingly.

LSST components interact via data shared by means of publish-subscribe communications using a Software Abstraction Layer (SAL)[12], a custom layer on top of an implementation of the Data Distribution Service (DDS) specification.[13]

The recommended implementation for the state machines is the State Pattern design pattern, following best-in-class practice.[11, 3]

3.1 Types of event triggers

A reactive system responds to various types of events.[14, 15] **Figure 1** describes these.

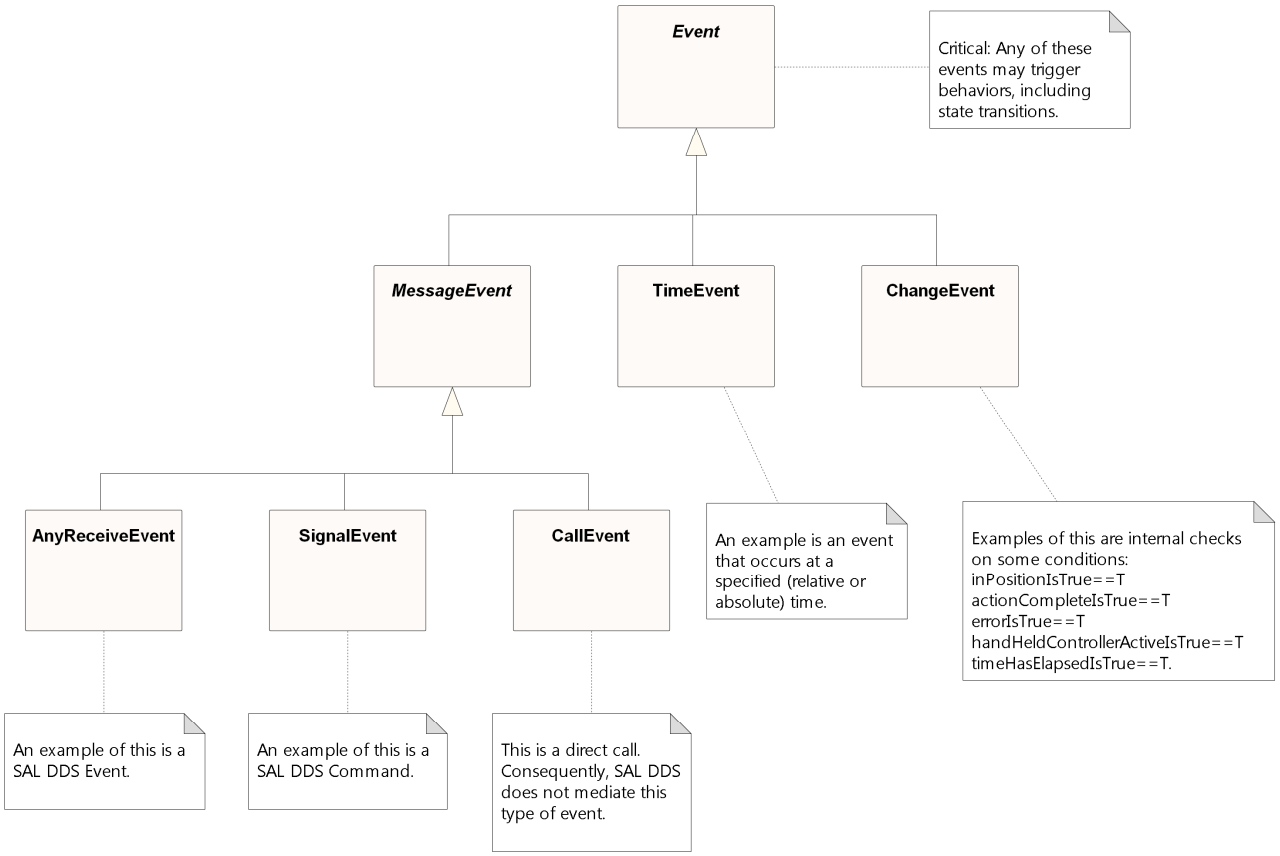

Figure 1. Types of event trigger

Of particular interest in this context are SAL DDS commands that clients to a component send as SignalEvents,

3.2 State definitions

Each system component has an agreed set of summary states. This references a common set of summary states defined in the Component Definition Base package.

Each system component extends this set of states to describe component-specific behaviors.

Figure 2 illustrates these relationships.

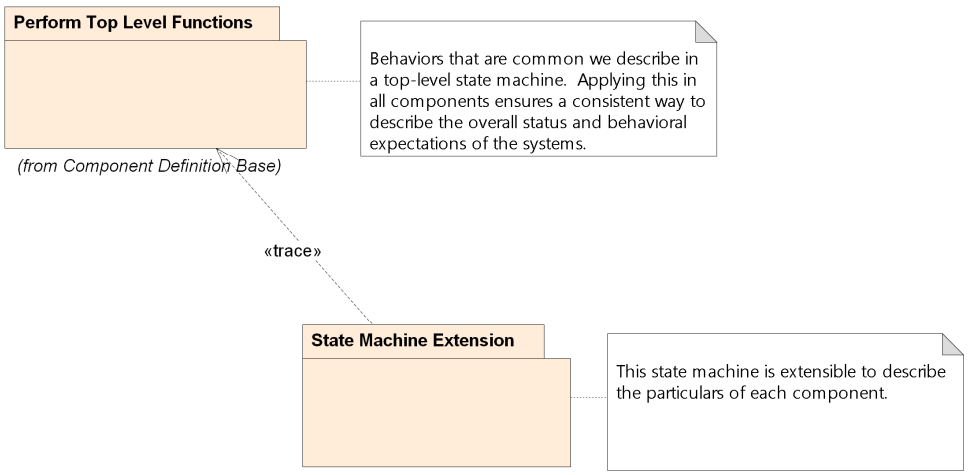

Figure 2. State machine extension

3.3 State machine commonality

Behaviors that are common we describe in a top-level state machine. Applying this in all components ensures a consistent way to describe the overall status and behavioral expectations of the systems.

3.4 State machine extension

We include some examples of modifying transitions to extend a state machine, either by changing actions on a simple transition or by overriding using composite states.to extend a state machine.[16] Compare the Ultimate Hook and Reminder patterns.[17]

A key concept here is run-to-completion.[16, 18]

3.5 Single step

One can extend a state machine by modifying actions defined on a single step transition (**Figure 3**).

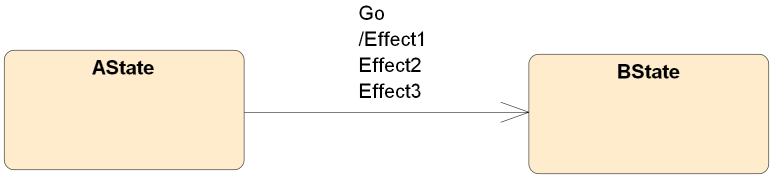

Figure 3. Single step

AState and BState are both simple states. All actions associated with the transition occur in a single, uninterruptible step. Modifying the actions associated with the transition redefines the behavior of the state machine.

3.6 Composite state

One can also redefine a simple transition with a composite transition (**Figure 4**).

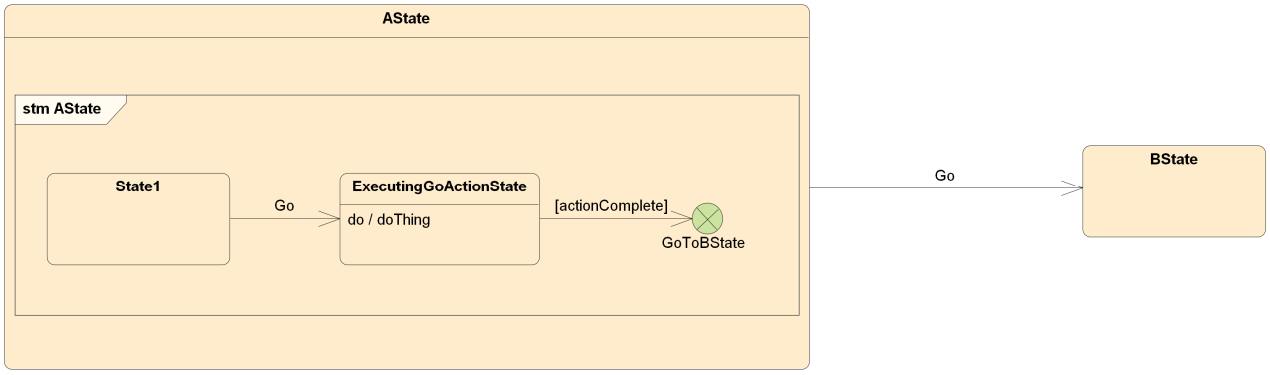

Figure 4. Composite state

In this example, the Go transition on State A is extended such that it does not complete in a single run-to-completion step, but uses composite states.

In this case, AState redefines the transition (overrides it) to include a transition to an intermediate state. A completion event ([actionComplete]) then triggers a transition to BState. In other words, there are now two triggers, each of which follows run-to-completion rules.

Note that:

1) There may be multiple steps.

2) The composite states may be in the source state, target state, or both. In order to maintain compliance with the summary state machine triggers, most often the composite states will be in the source state.

3.7 Component definition

This section presents a summary state machine (along with associated signals and type definitions) for a system component. This state machine serves as a base for extension, and we include two variations on the summary state machine itself.

For clarity, the state machine diagrams use the following convention to distinguish transition types:

triggerWithoutBrackets Indicates a SignalEvent.

[triggerInBrackets] Indicates a ChangeEvent. (Strictly speaking, this uses the Guard mechanism, without specifying a trigger. It is possible to mark a trigger formally as a ChangeEvent, but there is no visual distinction on a diagram between such a ChangeEvent and a SignalEvent. A Guard without trigger mechanism retains the logic of a ChangeEvent and provides a visual indication of the distinction on the diagram.)

The base summary states for a component and the transitions between these appear in **Figure 5**.

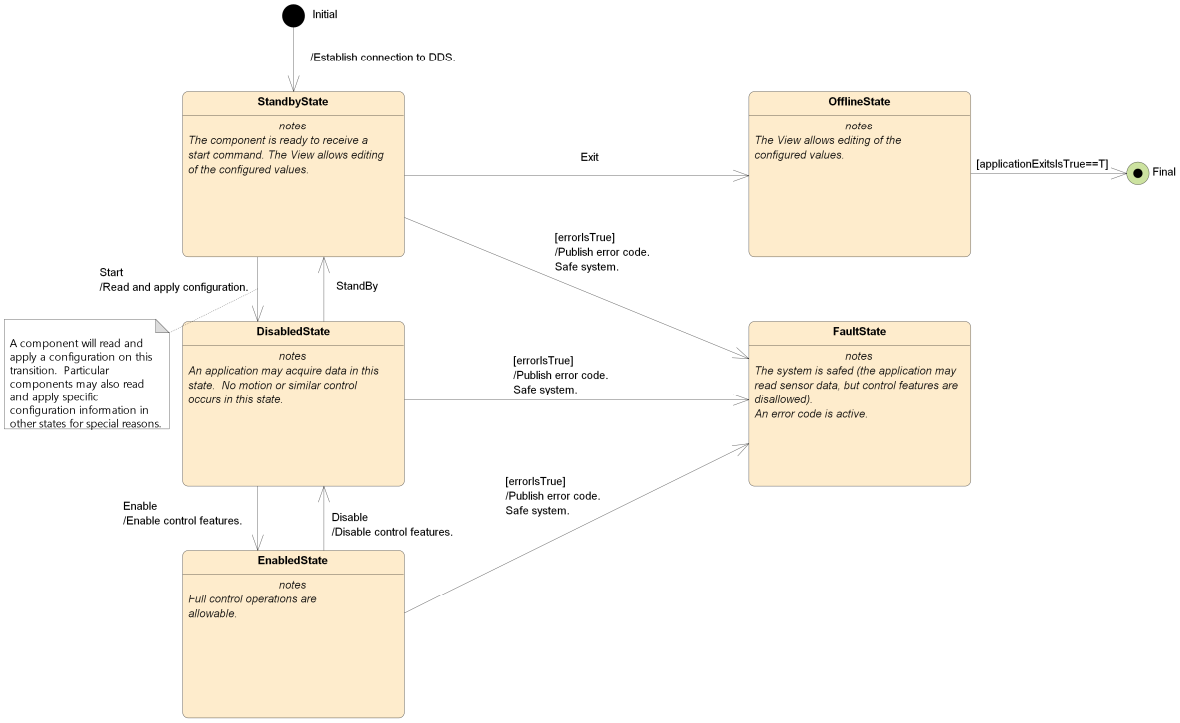

Figure 5. Perform top-level functions base

The Initial and Final pseudostates indicate the high-level entry and exit points for the state machine.

Initial Pseudostate: A component enters the scope of the model when it establishes a connection to DDS. The default model follows the simpler case when the component can accept commands through DDS as well as publish events and telemetry, so the initial transition is to a StandbyState.

StandbyState: The component is ready to start, and will do so upon receipt of a Start trigger.

Start Trigger: The component loads and applies the settings (specified as a command parameter) and transitions to a DisabledState.

DisabledState: A component has applied settings and may acquire data. No motion or similar control occurs in this state. Upon receipt of an Enable trigger, the component transitions to an EnabledState.

Enable Trigger: The component enables control features and transitions to an EnabledState.

EnabledState: Full control applications are allowable. Most components spend the majority of operating time in this state.

At times it will be necessary to send a Disable trigger to transition from EnabledState to DisabledState, disabling control features.

To apply different settings, a Standby transition takes the component back to StandbyState, where a Start transition may apply different settings. This feature makes it possible to switch all settings without going Offline.

From StandbyState, an Exit transition takes the component to OfflineState.

Exit transition: The component goes to OfflineState.

OfflineState: The component does not (generally) accept DDS commands.

When the component application exits, it transitions to the Final Pseudostate.

Final Pseudostate: The component becomes inaccessible to DDS and leaves the scope of the state machine.

If an error occurs, the component transitions to a FaultState, publishing an error code and making the system safe.

FaultState: The system is safe (the component may read sensor data, but control features are disallowed). An error code is active.

A component reads and applies a settings set on the Start transition. Particular components may also read and apply specific settings information in other states for special reasons.

Plans for future work includes adding a Stop internal transition in EnabledState.

Each of these summary states is a branch on a tree. An application may redefine the states and transitions through composite states or submachine states (using an orthogonal state machine).[19] Each application that does so will also report its present detailed state (i.e., leaf state).

3.8 Component definition extensions

LSST has defined an extension of the base state machine for components that use SAL for all communication with the controller, and another extension for components that use SAL for external clients of the controller and another communication mechanism for internal clients of the controller.

3.9 Component definition: SAL only commands

A first extension of the base summary state machine is for a component that uses SAL for all communication with the controller (**Figure 6**).

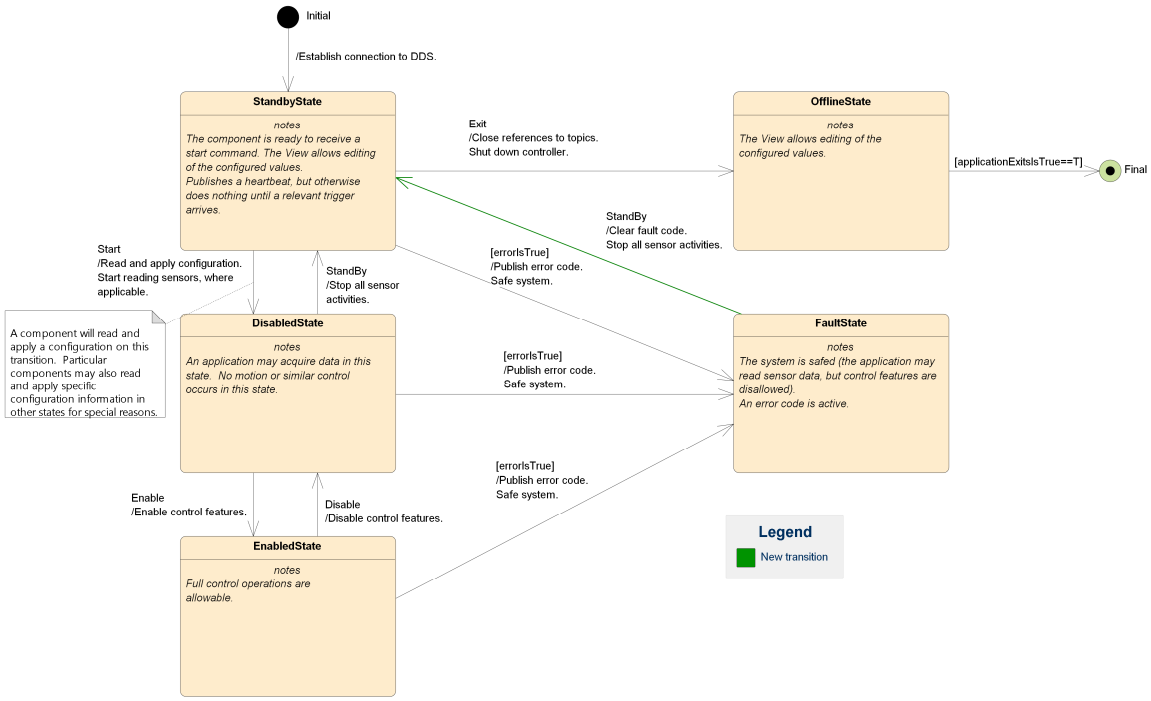

Figure 6. Extension for SAL only commands

This state machine extends the base state machine by adding a StandBy transition from FaultState to StandbyState.

In this extension, all commands to the component arrive as DDS messages. The component is not running already before it reaches StandbyState the first time.

Consequently,

In StandbyState, the component is publishing a heartbeat, but otherwise is essential quiescent until a relevant trigger arrives.

Consistent with this, on the Start transition, the component starts reading sensors. On the StandBy transition from DisabledState or FaultState, the component stops reading all sensors.

The Exit transition initiates a shutdown of the component. The component typically publishes it is in OfflineState and immediately exits.

3.10Component definition: SAL and other commands

A second extension of the base summary state machine is for a component that uses SAL for communication between an external client and the component controller, but another communication mechanism for communication between an internal client and the component controller (**Figure 7**).

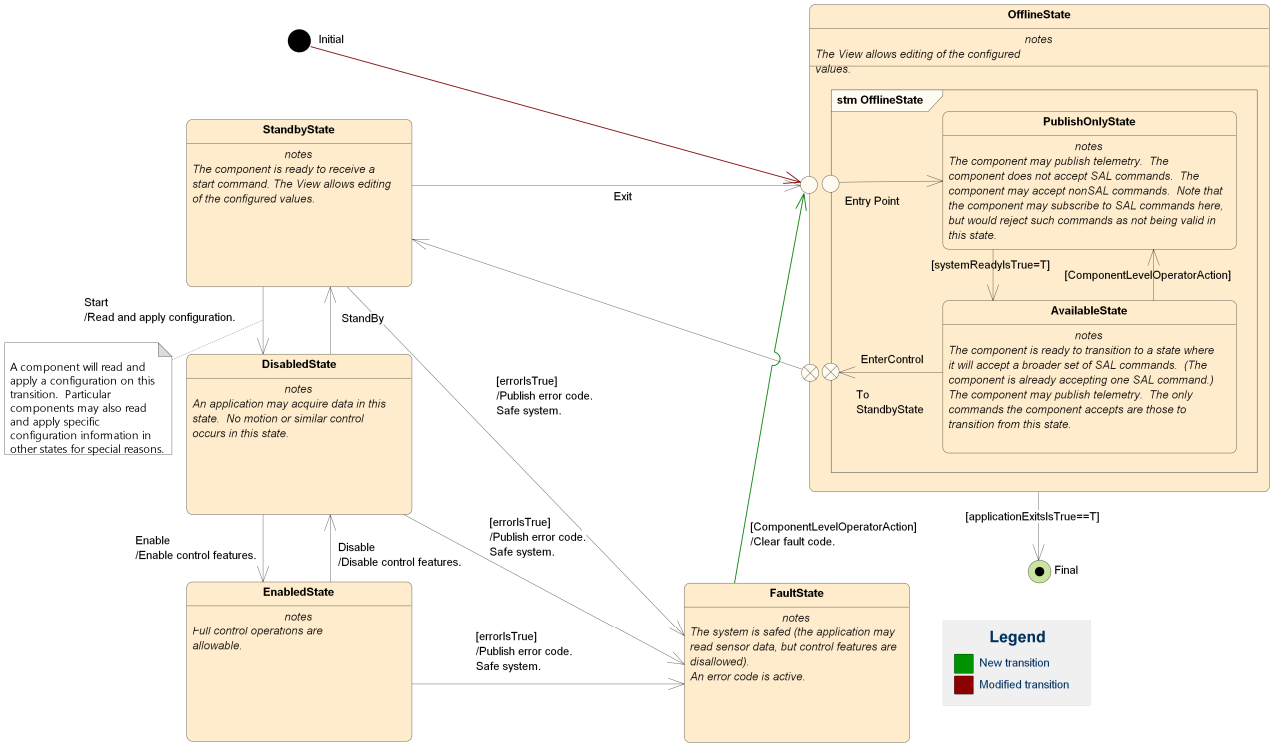

Figure 7. Extension for SAL and other commands

This state machine extends the base state machine by:

- Modifying the initial transition to go to a substate of OfflineState.
- Adding substates and associated transitions to Offline State.
- Adding a transition from FaultState to a substate of OfflineState.

Note that:

- A component of this type, once it establishes a DDS connection, starts under the control of its own nonDDS client. Hence it is Offline (it is not acting on SAL commands), even though it is powered on and functioning. It is connected to DDS and publishing state information. The control acitions possible in the PublishOnlyState are neither specified nor restricted by this model.

- Once the controller is ready, it is made available for control through DDS. Upon receipt of an EnterControl command, the component transitions to StandbyState. If a component was already reading sensors, it may continue to do so.
- On the Start transition, the component applies the requested settings. Settings are unknown or may vary through other mechanisms prior to this, so the component only has an official set of settings when it is in DisabledState, EnabledState, or FaultState.
- If an error occurs, the component still transitions to FaultState. This type of component, however, will incorporate a mechanism to go back to PublishOnlyState under OfflineState, where error resolution will occur using the component under nonDDS control.
- When the component application exits or disconnects from DDS communication, it transitions to the Final pseudostate, leaving the context of this diagram.

4. ENGINEERING FUNCTIONALITY

Various types of engineering functionality may apply to a system component. After characterizing the different types of engineering capabilities, we note how these might be implemented in a component controller or controllers.

4.1 Engineering capabilities

Different types of engineering activities typically occur (**Figure 8**).

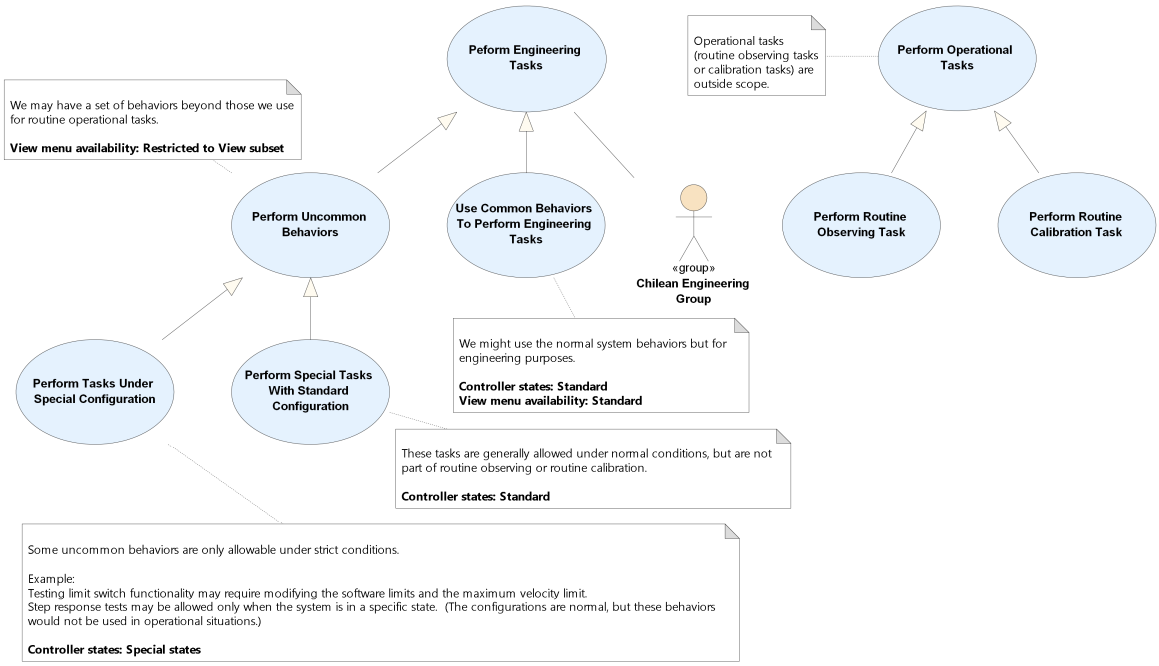

Figure 8. Engineering capabilities

Associated with the different types of activities are anticipated implications for a component view or controller.

4.2 Implementation: engineering and controllers

This section covers options (not necessarily exhaustive) to implement a particular subset of engineering functions (specifically those engineering functions that require special settings) in a component controller or controllers (**Figure 9**).

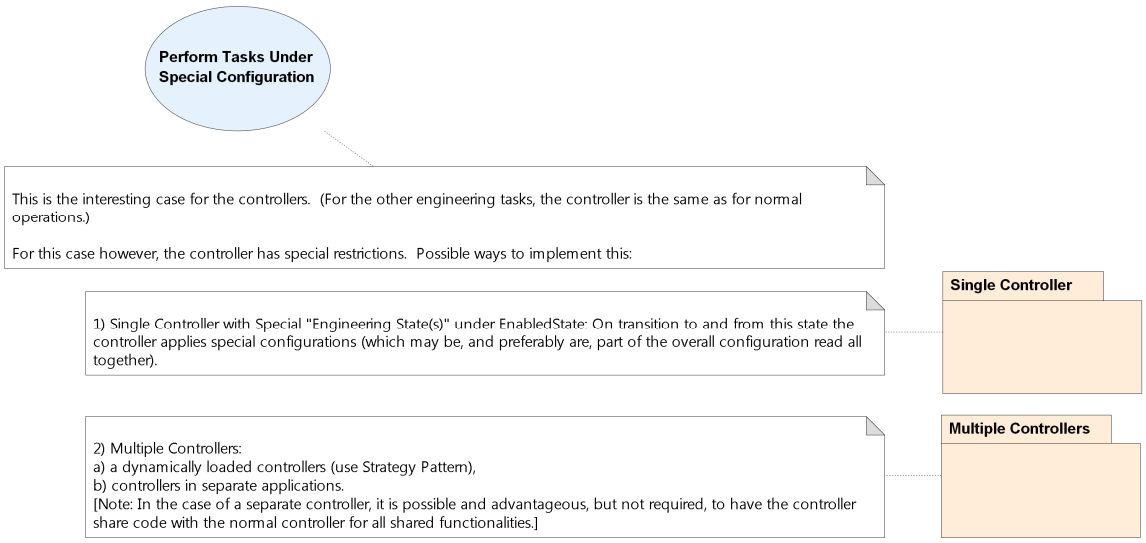

Figure 9. Engineering and controllers

For components that have tasks that require special settings, a single controller or multiple controller solution may apply.

4.3 Single controller

First we consider a Single Controller solution (**Figure 10**).

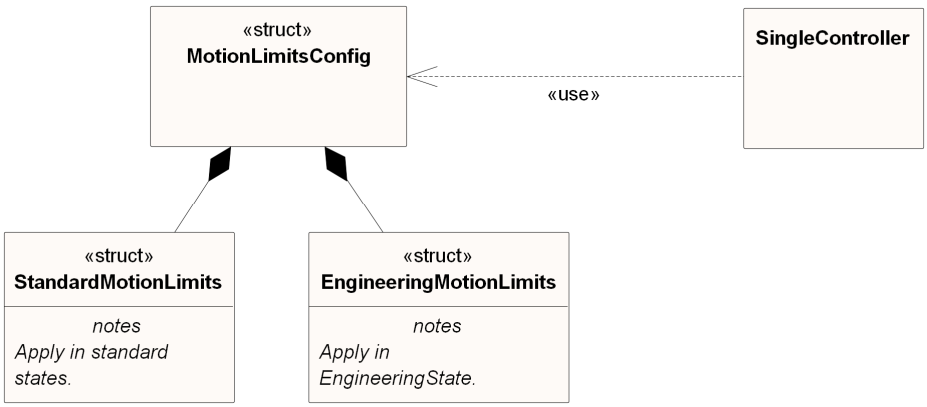

Figure 10. Single controller

The single controller applies the appropriate setting parameters as part of a state transition.

Differences in allowed triggers are handled as differences in state implementations.

4.4 Multiple controllers

Another option is a multiple controller solution.

A component may implement multiple controllers within a single application, or may allocate each controller to its own application (**Figure 11**).

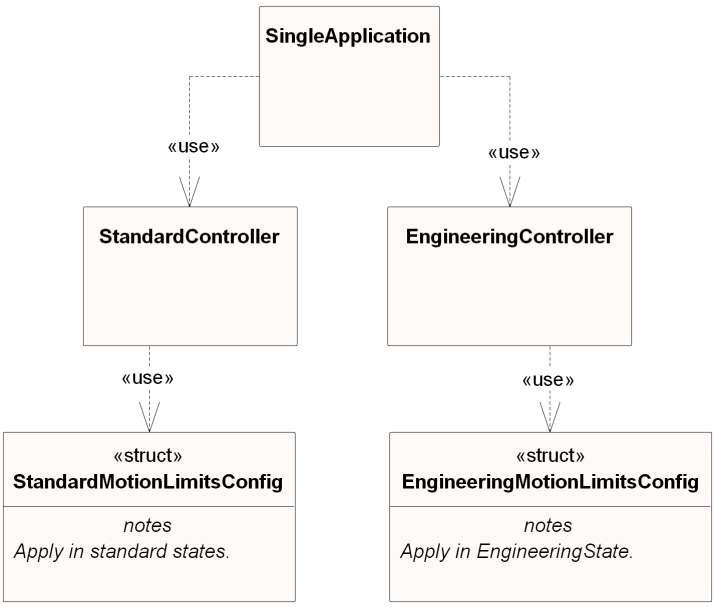

Figure 11. Single application

The application dynamically loads a separate controller, along with appropriate setting information, when transitioning from one mode or state to another.

Alternatively, the component may deploy the standard and engineering controllers in completely separate applications (**Figure 12**).

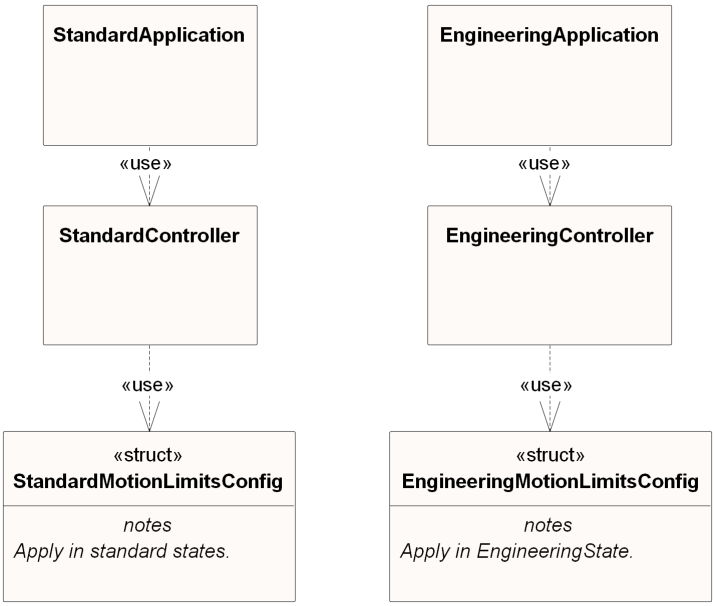

Figure 12. Multiple applications

5. SETTINGS

This section addresses how an LSST component manages settings. It includes related design definitions, structural descriptions, and examples.

5.1 Manage settings use cases

We have identified a set of use cases (which provide benefit to the user) for managing settings (**Figure 13**).

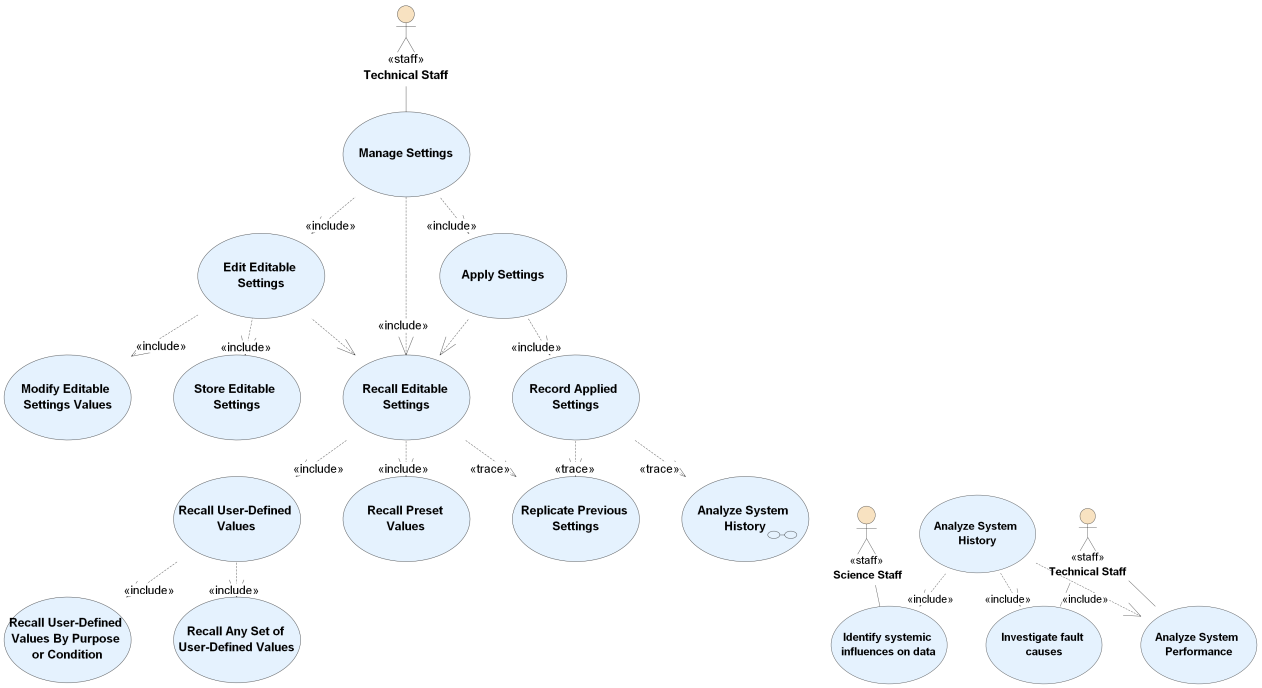

Figure 13. Manage settings use cases

Broadly speaking, managing settings involves recalling, editing, and applying settings. Both the Technical Staff and Science Staff derive benefits from being able to analyze which settings were applicable at any given time.

5.2 Settings design

A component manages its settings.

A user is able to edit editable settings. A component provides a mechanism for a user to change the values of editable settings (the full set only in the Standby or Offline States), which the component validates. Each component maintains its own store for setting values, and publishes a list of recommended versions of setting sets, upon connecting to SAL and when the versions in the store change.

The component applies a set of settings on demand (normally on the Start transition). The system records the settings applied.

The component recalls stored setting values on demand. These may be presets or values referenced using a version identifier or a label that maps to a specific version. The label mapping may change with time; the component publishes the mapping on start-up or when the mapping changes.

5.3 Setting definition

LSST provides a definition for a setting (**Figure 14**).

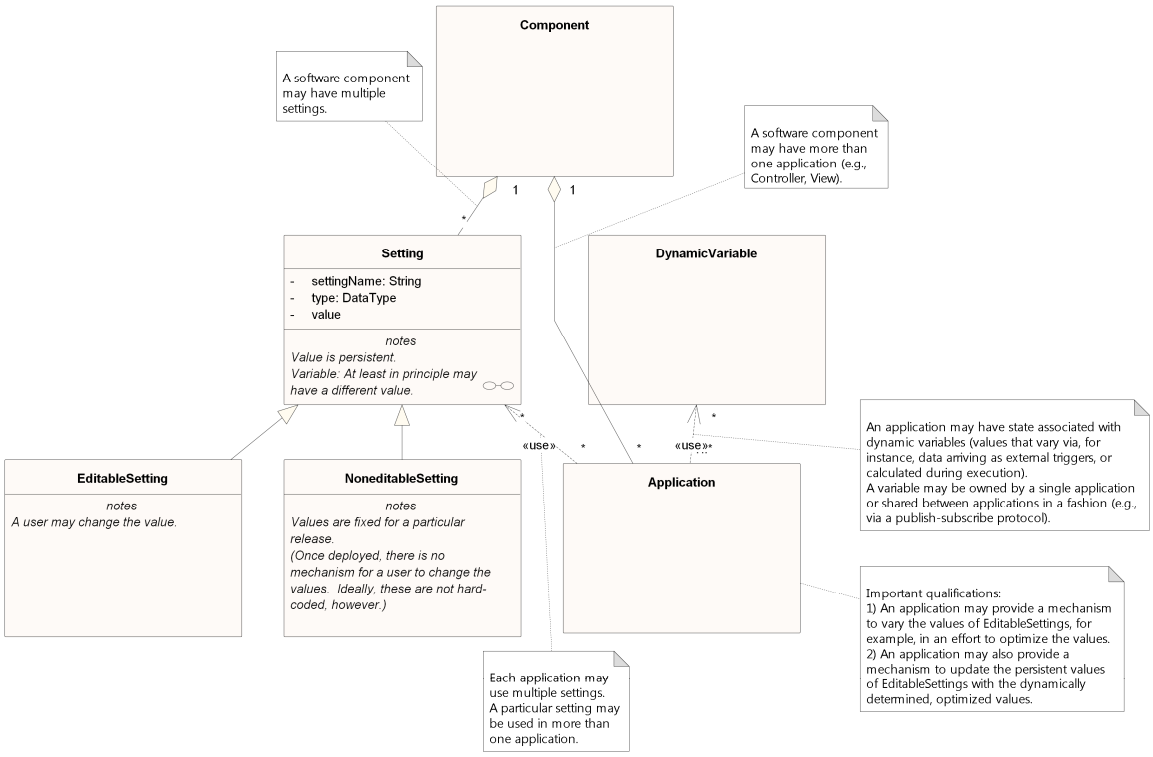

Figure 14. Setting definition

What is a setting?

- A setting is a variable that is persistent versus dynamic.
- A setting may be editable (value changeable by the user) or noneditable (value defined in a release).

A setting has a name, type (which may be a primitive or an array of primitives), and value (**Figure 15**).

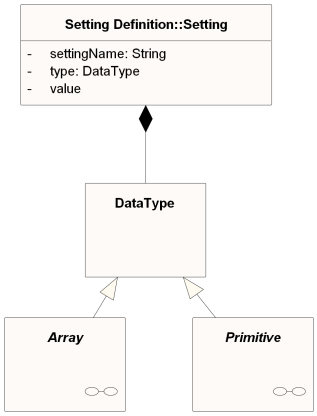

Figure 15. Setting structure

5.4 Settings hierarchy

This section covers the arrangement of settings into groups and sets, and the relationship to these of components and component applications (**Figure 16**).

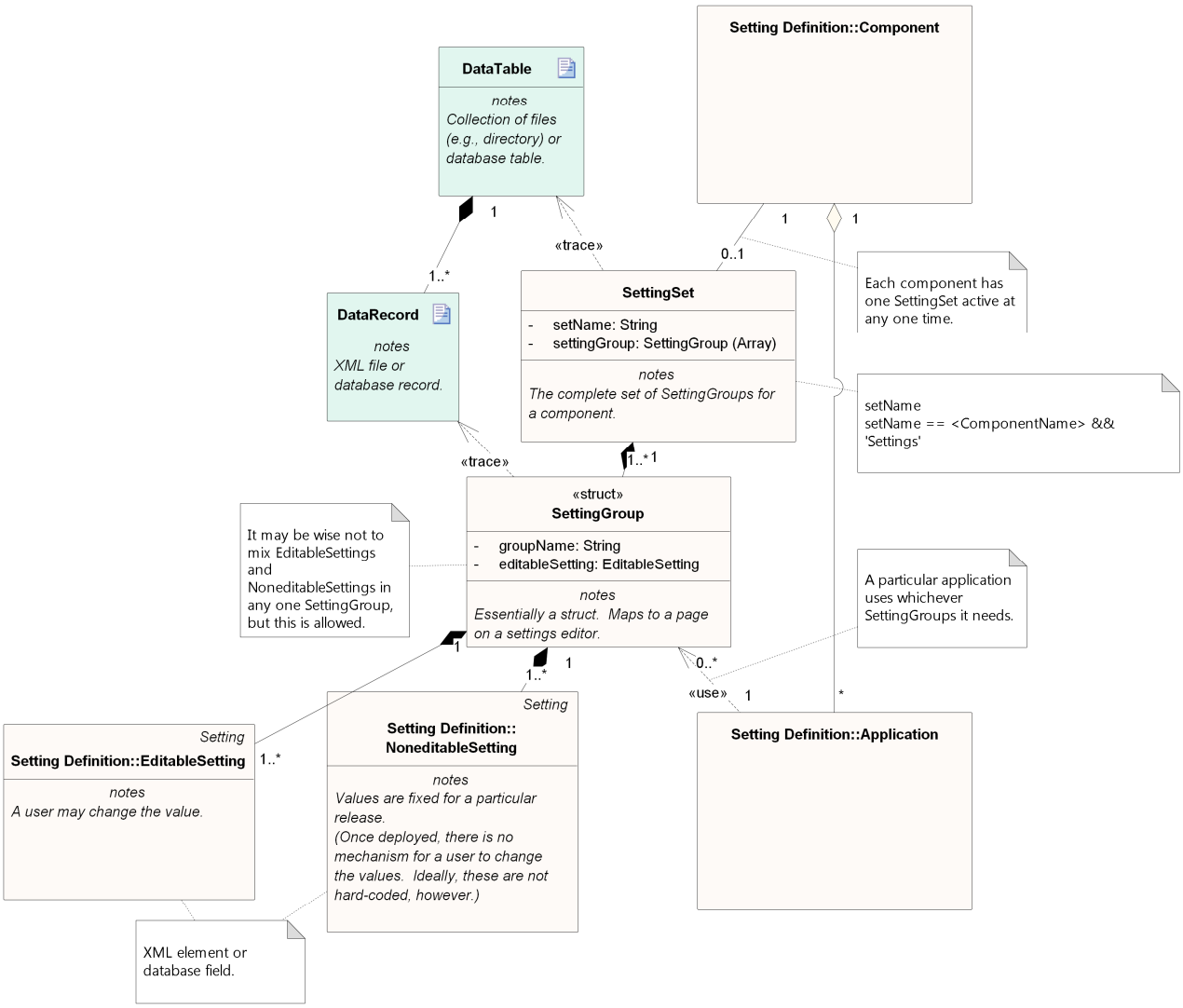

Figure 16. Settings hierarchy

We may logically arrange Settings into SettingGroups, which in turn we may assemble into a SettingSet, which includes all the Settings for a particular Component.

Closely related settings (those that always appear together) appear in a group.

The collection of all groups for a component define the complete set of settings for that component.

A particular application within a component may only need information for certain groups.

Note that a Setting may be editable or noneditable.

5.5 Settings hierarchy in editor

We show an example of how this may appear in a settings editor for a component (**Figure 17**). This example uses XML files to store groups of settings.

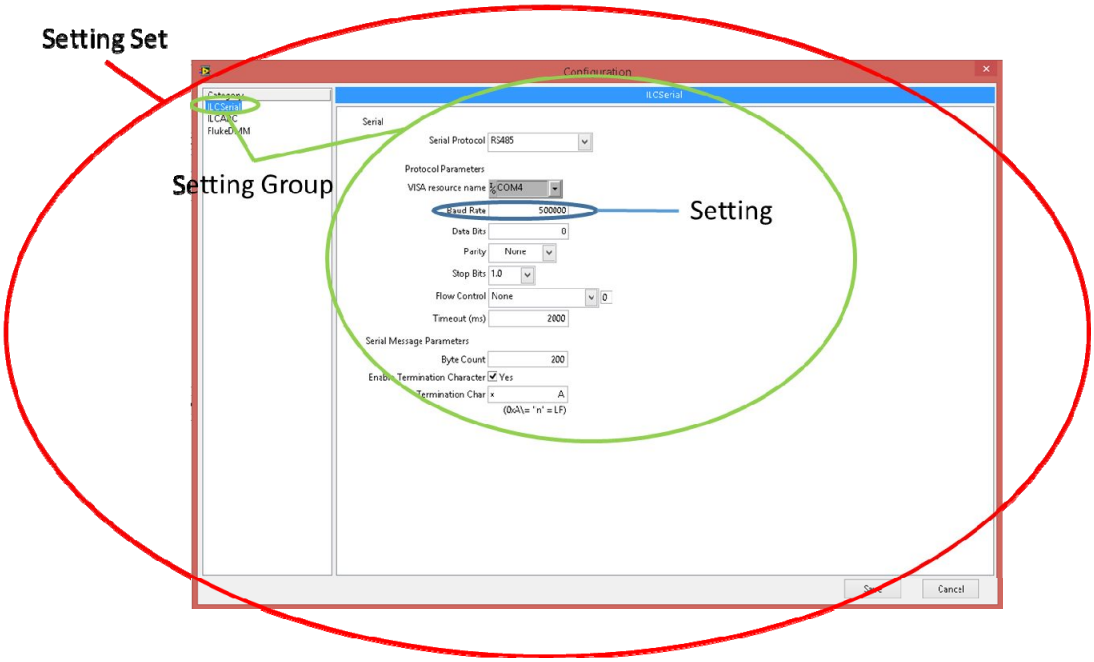

Figure 17. Settings hierarchy in editor example

A group of closed related settings maps to a page in the editor. The SettingSet is, then, the collection of all the settings on all the pages.

5.6 Settings hierarchy on disk

On disk, the settings hierarchy may follow the approach in **Figure 18**.

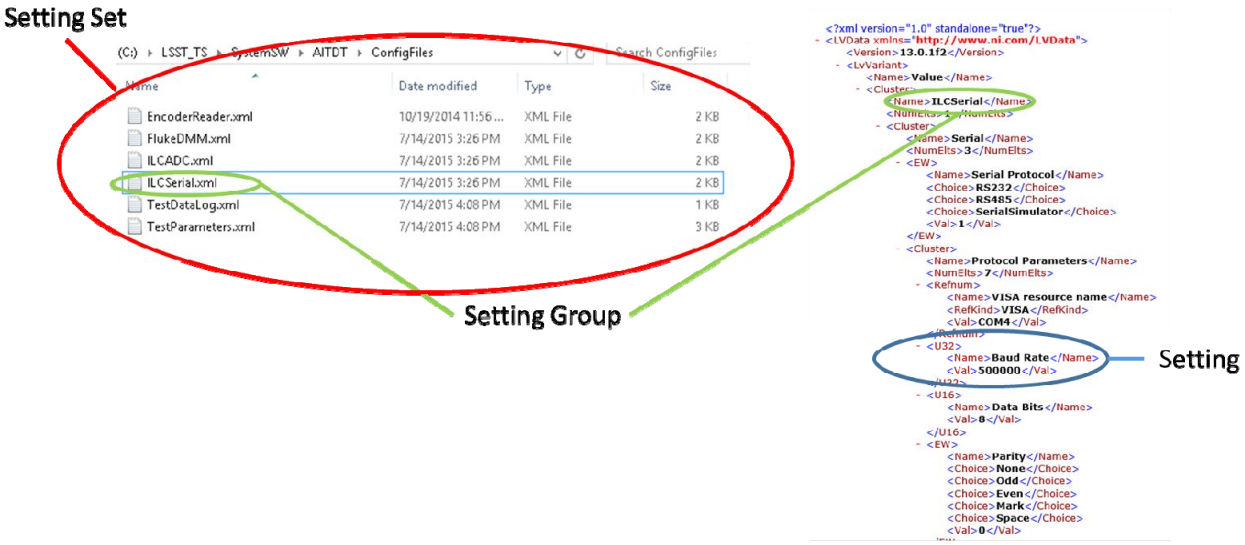

Figure 18. Setting hierarchy on disk example

There is an XML file for each SettingGroup. The collection of all XML files for a component corresponds to a SettingSet.

5.7 Settings store

The essential elements of a settings store appear in **Figure 19**.

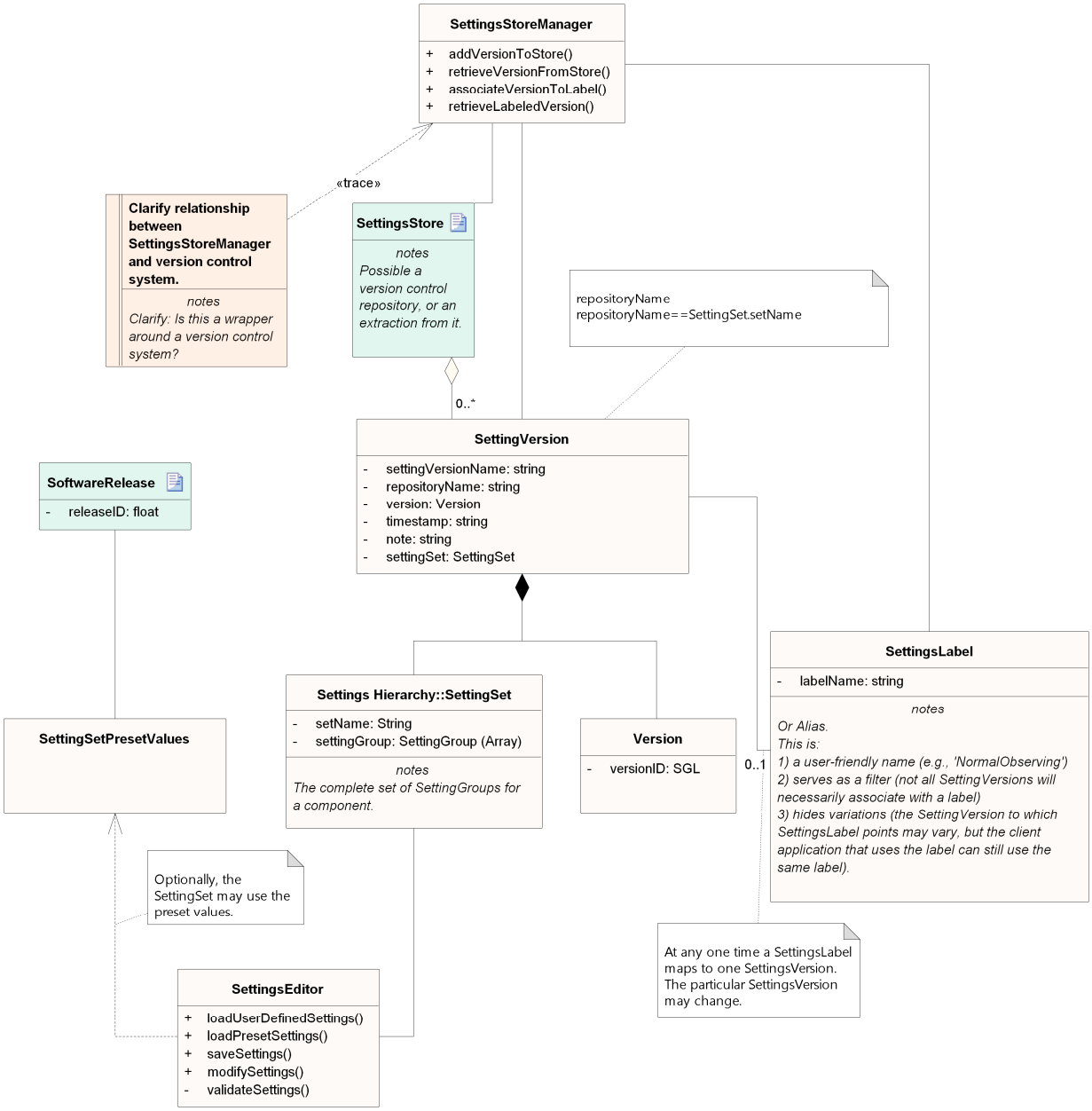

Figure 19. Settings store

A SettingsStoreManager maintains a Store of SettingVersions, each of which has a SettingSet (as modified with a SettingsEditor, for instance) and a Version.

A SettingsLabel maps to a SettingVersion. A user may change the particular version associated with a SettingsLabel.

The main concepts here are:

- A component manages versions of SettingSets.
- Each SettingSet has a version reference, which is fixed.
- A SettingsLabel provides an alternate reference to a SettingSet version. The particular version referenced by the label may change.

6. CONTROL REQUEST HANDLING

The system manages the network of commander-commandee relationships.

The key ideas are as follows:

- A person in the Site Manager role is able to edit and maintain white lists.
- The white lists define the network of which components may send commands to which other components.
- The SAL layer checks the white lists and only sends command messages that are allowed by the white list.
- The Site Manager or a proxy applies different white lists as the operational circumstances demand.

7. ALARM HANDLING

Components in the Telescope Control System (TCS) share a common way to handle alarm events (component behavior, display, and logging), which include warnings and faults. This approach may become a candidate for use in other subsystems, but this is a topic for future discussions.

7.1 Alarm Taxonomy

TCS components use the following terminology (**Figure 20**). Note that the terminology differs somewhat from that used in process industries (in, say, ANSI/ISA-18.2-2009 "Management of Alarm Systems for the Process Industries").

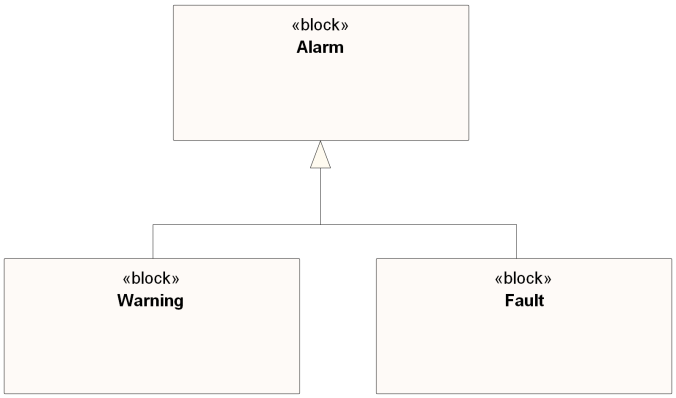

Figure 20. Alarm taxonomy

An alarm:

- is an event that indicates a condition that may impact component function or performance, and
- consequently deserves the attention of the user.

A warning represents an event that does not require a component to stop operation. No immediate action is expected by the component or the user, although a user may decide to take action based on one or more warnings.

A fault represents an event that requires a component to discontinue fair weather operation and safe itself.

7.2 Warning behavior

A component alerts users when the status of an alarm changes.

Each warning reportable by a component has its own SAL event topic defined on the component. (Potential warnings are known at the time of design.) When a warning occurs, the system sets the value of the corresponding warning topic to True (Active).

Each warning has an indicator on the corresponding View (**Figure 21**).

Figure 21. Warning indicator (warning inactive)

An active warning has a yellow background inside a bordered indicator, with Boolean text indicating the warning and its state (**Figure 22**).

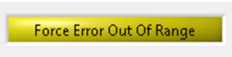

Figure 22. Active warning

The component view includes all warnings in a group in the summary area (**Figure 23**).

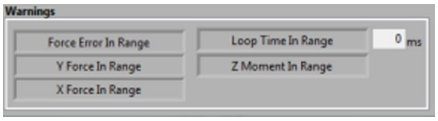

Figure 23. Warning group

An inactive warning has a transparent background inside a bordered indicator, with Boolean text indicating the warning and its state.

7.3 Warning summary display

Each system of components includes a warning summary display at the top-level View. An example of this would be a Warnings Tree Display (**Figure 24**).

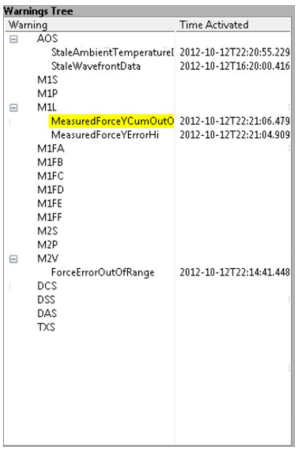

Figure 24. Warnings tree display

The warning summary display only lists active alarms, indicates the activation time for each active warning, and highlights the most recently activated warning. (Note that, if the most recent warning clears, no warning is highlighted.)

The warning summary display displays active warnings in the component to which they originate. The component settings permit customization of which warnings from which components are included. The default settings include warnings for the component and its children.

If the warning clears, the system resets the value of the corresponding warning topic to False (Inactive). When a component starts, it sets all warnings to Inactive. (An additional history display is welcome.)

7.4 Fault behavior

If an abnormal event occurs that requires the component to stop normal operation, the system handles this as an error.

Each component has a single ErrorCode topic. When a fault occurs for a component, the component publishes an appropriate error code to the component error code topic.

The component defines each custom error code and its associated text in an error code file associated with the component. Each component defines error codes within the range allocated to it.

If an error is active, the component View displays the active error code and its associated description (**Figure 25**). The suggested practice is that this display is visible only when an error is active.

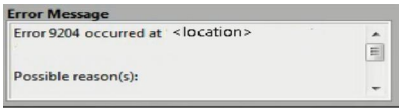

Figure 25. Error code display

When a fault occurs for a component, the component transitions to FaultState, publishing the state change. The component safes itself (stop operations, such as motion, that pose equipment risks; safe operations such as data acquisition may continue) gracefully on the transition to FaultState.

A component permits resolution of a fault at the component with the fault condition.

Each fault requires action by a user to reset (clear). In most cases there will be global method. (It is generally not possible to know all possible faults in advance.) Specific faults may require user acknowledgment of the specific fault before a global reset will take effect.

If a user resets the fault, the component publishes error code 0 as the active error code. When a component starts, it resets the error code to 0.

If a user resets a fault, the component transitions out of FaultState.

A component handles faults at the appropriate code layer. The reference implementation is to use the State Pattern along with a periodic interrupt ("Update") event. While processing each trigger, the application creates an error group where an error condition occurs. At the end of "Update" trigger processing, in the <SummaryState> method, the code handles the case when error==True.

A component that is a parent displays on its View its own summary state and the summary states of its children in the status section. (This does not include fault codes.) See **Figure 26**.

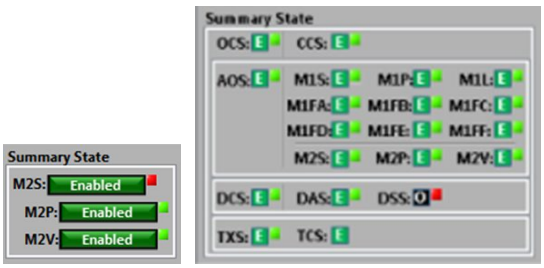

Figure 26. Summary state indicator hierarchy in full and abbreviated forms

The summary state indicator has a red background when indicating the component is in FaultState (**Figure 27**).

Figure 27. Summary state indicator in FaultState

7.5 Alarm logging

Each component log s all warning and fault events. Note that it is sufficient to publish SAL events as described earlier to satisfy the requirement. The system already has functionality to log SAL events.

8. CONCLUSION

We presented common design elements for LSST system components. Each component shares a summary state machine that includes a methodology for handling engineering tasks. Each component manages its own settings. A control request handler facilitates command-commandee relationships. Components report and handle alarms. Common approaches to these will help ensure successful interaction between components, robust component design, and a uniform look and feel for users.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation through Cooperative Support Agreement (CSA) Award No. AST-1227061 under Governing Cooperative Agreement 1258333 managed by the Association of Universities for Research in Astronomy (AURA), and the Department of Energy under Contract No. DEAC02-76SF00515 with the SLAC National Accelerator Laboratory. Additional LSST funding comes from private donations, grants to universities, and in-kind support from LSSTC Institutional Members.

We thank Stuart Marshall, SLAC/LSST for reviewing the manuscript.

REFERENCES

[1] Kahn, S., “Final design of the Large Synoptic Survey Telescope,” in [Ground-Based and Airborne Telescopes IV], Hall, H. J., Gilmozzi, R., and Marshall, H. K., eds., Proc. SPIE 9906, in press (2016).

[2] Lotz, P. J., Lacasse, M. J., and Godwin, R.C., "Discovery Channel Telescope software component template and state design: principles and implementation", in [Software and Cyberinfrastructure for Astronomy II], Radziwill, N.M., Chiozzi, G., eds., Proc. SPIE 8451, 845108 (2012).

[3] Lotz, P. J., “State pattern implementation for scalable control systems,” NI Week TS 8237 (2012). https://decibel.ni.com/content/docs/DOC-23603.

[4] Lotz, P.J., Lacasse, M. J., and Godwin, R. C., “System components using design patterns on the Discovery Channel Telescope,” NI Week TS 8245 (2012). https://decibel.ni.com/content/docs/DOC-23606.

[5] Lotz, P.J. and Lacasse, M.J., “Creating system components with object-oriented programming,” NI Developer Day, Phoenix (2011). https://decibel.ni.com/content/docs/DOC-15913.

[6] Lotz, P. J. and Lacasse, M. J., “System components with object-oriented design patterns,” NI Week TS 1665 (2011). https://decibel.ni.com/content/docs/DOC-17376.

[7] Weilkiens, T., [Systems Engineering with SysML/UML: Modeling, Analysis, Design], Morgan-Kaufmann, Amsterdam, 8 (2006).

[8] Black, P. E., "state machine", in Dictionary of Algorithms and Data Structures [online], Pieterse, V. and. Black, P. E., eds. 12 December 2005. (Accessed 22 May 2016) Available from: http://www.nist.gov/dads/HTML/statemachine.html.

[9] [OMG Unified Modeling Language, Superstructure, Version 2.4.1], Object Management Group, 535 (2011).

[10] Weilkiens, T., & Oestereich, B., [UML 2 certification guide: Fundamental and intermediate exams], Elsevier, Amsterdam, 224 (2007).

[11] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., [Design patterns; elements of reusable object-oriented software], Addison-Wesley, Boston (1994).

[12] Mills, D., “LSST communications middleware implementation,” in in [Ground-Based and Airborne Telescopes IV], Hall, H. J., Gilmozzi, R., and Marshall, H. K., eds., Proc. SPIE 9906, in press (2016).

[13] Object management Group, “DDS Specification”, http://www.omg/org/spec/DDS/1.4 (2015).

[14] Weilkiens, T., & Oestereich, B., [UML 2 certification guide: Fundamental and intermediate exams], Elsevier, Amsterdam, 146 (2007).

[15] [OMG Unified Modeling Language, Superstructure, Version 2.4.1], Object Management Group, 443 (2011).

[16] [OMG Unified Modeling Language, Superstructure, Version 2.4.1], Object Management Group, 15.3.12 (2011).

[17] Samek, M., [Practical UML statecharts in C/C event-driven programming for embedded systems, (2nd ed.)], Newnes/Elsevier, Amsterdam, Ch.5 (2009).

[18] Samek, M., [Practical UML statecharts in C/C event-driven programming for embedded systems, (2nd ed.)], Newnes/Elsevier, Amsterdam, 67-68 (2009).
[19] [OMG Unified Modeling Language, Superstructure, Version 2.4.1], Object Management Group, 15.3.11 (2011).