Project Documentation
Document SPEC-0014
Revision B

Software Design Document

Steve Wampler, Bret Goodrich
Software Group

23 April 2009

Advanced Technology Solar Telescope 950 N. Cherry Avenue Tucson, AZ 85719
Phone 520-318-8102 atst@nso.edu http://atst.nso.edu Fax 520-318-8500

REVISION SUMMARY:

1. Date: Aug 7, 2003
   Revision: A1
   Changes: Initial version
2. Date: Aug 8, 2003
   Revision: A2
   Changes: Minor modifications
3. Date: Aug 8, 2003
   Revision: A
   Changes: Additional of CAR description, more on configurations
4. Date: July 26, 2004
   Revisions: B1
   Changes: Converted to HTML for TWiki
5. Date: Aug 24, 2006
   Revision: B2
   Changes: Adjusted for document renaming
6. Date: 12 September 2006
   Revision: B3
   Changes: Renumbered revisions per standard. Minor formatting changes.
7. Date: 23 April 2009
   Revision: B
   Changes: Updated for FDR

Table of Contents

Preface

This document provides an overview of the design of the ATST software systems. It is a living document that is expected to evolve throughout the design process. During conceptual design it provides a 'broad-brush' perspective of the design with detail to be added during subsequent design phases. The focus during conceptual design is on describing enough of the design to allow an examination of the design's suitability in meeting the system requirements. In this fashion, the document presents many conceptual design concepts as design requirements.

Tasks are broken out into development areas suitable for allocating to development teams. Areas of significant risk are identified and measures to be taken to address those risks are discussed. While system interfaces are identified, they are not detailed until the preliminary design process; examples used here should be taken as illustrative only at this point in the design process. Where appropriate, baseline choices are indicated. These should be viewed more as discussion aids at this point, though comments on the perceived suitability of these choices are welcome.

The presentation of the design itself follows the structure presented in early documents: the design of the system as it relates to the functional architecture is presented first, followed by those design considerations drawn from the technical architecture. Because the functional architecture closely follows similar architectures found in other modern telescopes, the initial concentration is on the technical design, which is more interesting during conceptual review.

The ATST Software Design Document (SDD) follows from the Software and Controls Requirements (SCR) and the ATST Software Concepts Definition Document (SCDD); readers should be familiar with at least the terminology introduced in those documents. The SDD serves as a reference for the various subsystem requirements documents. The role within the project of the SDD is to act as a reference against which the system implementation is measured. The Observatory Control System Specification Document, Telescope Control System Design Requirements Document, Instruments Control System Specification Document, and the Data Handling System Specification Document are companion documents providing the software design requirements specific to each of the principal systems.

1. INTRODUCTION

1.1 GOALS AND REQUIREMENTS

The model presented in this document is intended to address the following goals and requirements:

- Flexibility in a laboratory style operating environment
- Common control behavior to simplify operations
- Reduced development costs through reuse and separation of functional and technical architectures
- Reduced integration costs
- Reduced maintenance costs
- Simplification of instrument setup/take-down to reduce the chance for errors and increase observing efficiency.

1.2 ACRONYMS

- ACS—ALMA Common Software
- ALMA—Atacama Large Millimeter Array
- ATST—Advanced Technology Solar Telescope
- CA—EPICS' Channel Access
- CSF – ATST Common Services Framework
- CORBA—Common Object Request Broker Architecture
- DHS—Data Handling System
- DST—Dunn Solar Telescope
- EPICS—Experimental Physics Instrument Control System
- ESO—European Southern Observatory
- ICE—Internet Communications Engine
- ICS—Instrument Control System
- OCS—Observatory Control System
- NDDS—Network Data Delivery Service
- SCA—SOAR Communications Architecture
- SQL—Structured Query Language
- TBD—To Be Determined
- TCS—Telescope Control System
- XML—Extensible Markup Language

1.3 GLOSSARY

- CSF—ATST Common Services Framework: software provided as part of the ATST control system for use by all system components.
- Client—a software entity that makes requests of other software entities (called servers).
- Component—a software entity with formal interfaces and explicit dependencies. Components must be remotely accessible in a distributed environment.
- Container—part of the CSF, containers insulate Components and Controllers from their physical location and provide core services to those Components.
- Controller—a Component that follows the command-action-response model. Some components connect to physical devices.
- Functional architecture—a view of the software architecture that focuses on system behavior and component interactions
- Functional interfaces—those interfaces that characterize the functional behavior of specific Components
- Life cycle interface—the interface allowing control of a Component's life cycle: starting, stopping, relocating, monitoring, etc.
- Module—the implementation of a Component or Controller within a container
- RDBMS—a Relational Database Management System
- Server—a software entity that responds to requests from other software entities (called clients)
- Services interfaces—interfaces used by Components to access CSF services
- Technical architecture—a view of the software architecture that focuses on the implementation and support structures

1.4 REFERENCED DOCUMENTS

1. SPEC-0001, *Science Requirements Document*.
2. SPEC-0005, *Software and Controls Requirements*.
3. SPEC-0013, *Software Concepts Definition Document*.
4. SPEC-0015 , *The Observatory Control System Specification Document*.
5. SPEC-0016, *The Data Handling System Specification Document*.
6. SPEC-0022-1, *The Common Services Design Requirements Document*.
7. SPEC-0019, *Telescope Control System Design Requirements*.

2. DESIGN OVERVIEW

2.1 SYSTEM OVERVIEW

The Advanced Technology Solar Telescope (ATST) software system is a highly distributable client-server system organized into four co-operating peer systems: the Observatory Control System (OCS), the Telescope Control System (TCS), the Instruments Control System (ICS) and the Data Handling System (DHS) as shown in figure 1. A communications bus serves as the backplane for communications. This communications bus is discussed in detail later.

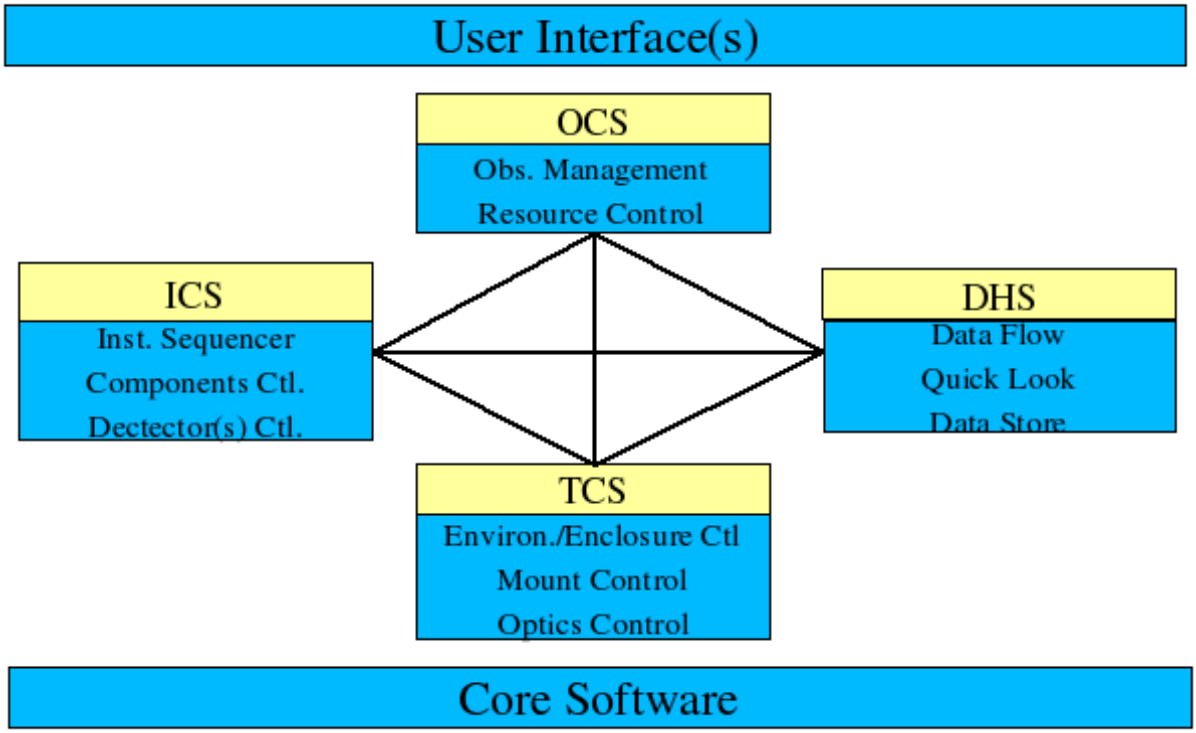

Figure 1 - Major ATST software systems

Each of these major systems is responsible for different aspects of observatory operations.

2.1.1 Observatory Control System

The OCS is responsible for high level observatory operations including user interfaces, scheduling, resource allocation, and system monitoring and maintenance. In addition, the OCS development team is responsible for the ATST Common Services Framework (CSF), which includes the design and implementation of the communications bus and other core software.

The OCS is the 'control panel' through which users access the ATST and interacts closely with system operators. While the design allows for later increased levels of automation, initial efforts are on providing a suitable environment for operator control.

2.1.2 Telescope Control System

The TCS provides coordination and control of the various telescope components to deliver high quality images to the instrument stations. It provides capabilities for target acquisition and tracking, image quality and location, and thermal management.

The TCS is directly responsible for ephemeris calculations, pointing, target trajectories, blending of image quality data, coordination of observations, and safety interlocks. It also controls the following subsystems to meet its requirements:

- The Enclosure Control System (ECS) operates the enclosure and associated hardware (shutter, ventilation, thermal control).
- The Mount Control System (MCS) drives the mount and coudé rotator.
- The M1 Control System (M1CS) shapes the primary mirror figure.
- The M2 Control System (M2CS) operates the secondary mirror.
- The Feed Optics Control System (FOCS) drives the feed optics.
- The Adaptive Optics Control System (AOCS) provides high-quality images.
- The Global Interlock System (GIS) controls the safety interlocks.

2.1.3 Instrument Control System

The ICS provides a common framework for all planned ATST instruments, both facility class and visitor. Because of the highly reconfigurable requirements of most solar observations, the ICS manages a set of instruments. The instrument set provides both the necessary communications infrastructure an instrument needs and a flexible management system to perform experiments involving the simultaneous, coordinated use of multiple instruments.

2.1.4 Data Handling System

The DHS has three initial responsibilities:

- Data flow support for scientific data collection,
- Quick look support for quality analysis and control, and
- Distribution of scientific data.

The design allows for adding additional functionality such as a data processing pipeline, data archiving and retrieval, etc.

The largest factor driving DHS design is the potential data rates proposed for scientific data collection. Since some of those data rates exceed current network and backplane technologies, the design pushes initial data reduction steps into the instruments themselves.

2.1.5 Common services framework

Most observatories now recognize that both development and operational costs are affected by the software architecture. These costs can be reduced by adopting, as much as feasible, a common framework for all software. In addition, the need to combine components into new instruments in a laboratory-style setting also indicates an advantage in such a common framework.

By providing a common services framework, ATST can avoid duplicating effort and help ensure a consistent behavior among systems. Subsystem developers can focus on providing their required

functionality and not on developing the structure required to integrate that functionality into the ATST software system.

The common services framework includes support for the communications bus. The communications bus is provided by COTS middleware (ICE is the baseline choice) and allows peer-to-peer communication between Components without regard to physical location. The communications bus also provides uniform implementation of the communications protocol layers up through the interface layer. This includes standard interfaces for all system services.

Other features provided by the Common Services Framework include:

- Implementation of Containers
- Support for Component implementation, including access to system services
- Templates for common Components
- The specification and standard template implementing the Controller and command-action-response model.

3. FUNCTIONAL DESIGN

All of the major systems share a common overall structure. All are designed to be distributable and communicate externally and internally across the communications bus. Each has a Master Application responsible for starting, stopping, and monitoring the overall condition of that system's components. Each has a Components Manager for managing the operation of specific components found in that system and its subsystems. The Master Application and Components Manager access components that provide system specific functionality. These and other aspects of the common structure are discussed under the subsection Observatory Control System since that system is responsible for managing the common services framework.

3.1 INFORMATION FLOW

Control and status information flows through the ATST control system using a message passing architecture. The types of messaging are:

- Commands
- Events
- Persistent store access
- Alarms
- Log messages

3.1.1 Commands

Commands are performed using the Command/Action/Response model used in Gemini and SOLIS. The fundamental entity for command processing is the configuration. A configuration contains all the information necessary to move a system component from one state to another. The information in a configuration is represented by a set of attribute/value pairs. All configurations in ATST are uniquely identified and all Components track the configuration on which they are currently operating.

Components respond to configurations in one three ways:

- No response is needed. Some systems and configurations operate in a stateless manner. For example, the logging system simply accepts messages and records them no response is required.
- An immediate action is performed and acknowledged. If the action can be performed effectively instantaneously (recording information in a local variable might be sufficient action for some configurations [setting the debug level of a Component is an example]), the command response provides the result of performing that action.
- The configuration is acknowledged as valid and an action is initiated. When the action completes a separate mechanism (callback or event) announces the completion.

As actions are being performed by a Component in response to a command, the Component may receive and generate events, make requests of the persistent stores, generate alarms and record log messages.

3.1.2 Events

Events are used for asynchronous message delivery. The event system in ATST is modeled after conventional notification services. Events are a publish/subscribe mechanism that is based on event names: Components may post events without regard to the recipients and subscribe to events without

regard to their source. Subscriptions may be made using wild-carded names to subscribe to multiple events sharing common name characteristics.

3.1.3 Persistent store access

Components may store and retrieve information from a persistent store. Configuration parameters for Components are maintained in the persistent store and are retrieved by Components on initialization. Similarly, Components may save checkpoint parameters into the persistent store if needed.

The persistent store is implemented using one or more databases permitting searchable access to the contents. This allows other system Components to obtain (and, in the case of managers, update) configuration parameters.

Databases are also used to hold header information, log messages, and configuration information.

3.1.4 Alarms

Alarms are propagated using the event system. Components may generate alarms, respond to alarms, and recast alarms.

3.1.5 Log messages

Components may generate messages that are recorded into a persistent store.

3.2 OBSERVATORY CONTROL SYSTEM

The OCS provides both services and software support that are available for use by all principal systems:

3.2.1 User Interfaces

One of the key tasks of the OCS is to provide user interfaces (UI) for human interaction with the ATST software during normal operations. Traditionally, UI development is one whose effort is consistently underestimated, in part because the expectations of users change rapidly as the system design progresses. This results in many rewrites and redesigns of UI. Consequently OCS design of UI support is based on the precept that flexibility is fundamental to UI development. For similar reasons, GUIs are developed using the Model/View/Controller (MVC) model.

User interfaces are separated from control operations in the OCS. Control components provide a common control surface that is used by Graphical User Interfaces, scripts, and programmatic access. These control surfaces are available through the communications bus. This allows UI applications to be distributed independently of the location of the underlying control components. Further, since all user interfaces (GUI, script, programmatic) operate through the same control surface, any form of user interface can be used to access any control surface. This means that simple UI may be produced quickly to aid in development and safely replaced with more sophisticated UI later. Status information is also provided through common interfaces across the communications bus allowing similar flexibility in UI display of status information.

Because of the flexibility of this approach, little is presented here on UI specifics. Instead, it is noted that any component that presents a common control surface and common status interfaces may have zero or more UI attached to it.

User interfaces are *composable*. User interfaces are implemented as classes (or the equivalent) and can be combined to form UI applications. This allows sophisticated UI to be built quickly from simpler base UI components.

Engineering interfaces, while the responsibility of system developers, are implemented using the APIs and libraries provided by OCS.

While there are no plans to provide web-based UI, the separation of UI and control makes this a relatively straightforward task should web-based control be required in the future.

3.2.2 System management

The OCS is responsible for overall system management. Through the OCS, the system can be started, monitored and adjusted during operation, and stopped. The OCS Master Application functions as the ATST Master Application and is responsible for managing these activities across ATST. However, actual control is delegated to the Master Applications in each of the major systems.

3.2.3 Component management

The fundamental building block of the ATST distributed architecture is the Component, discussed in more detail in the Technical Design section below. For the discussion here, it is sufficient to note that applications are composed of one or more Components. Component management refers to the process of Component life cycle and configuration management.

ATST recognizes three different Component life cycle models:

- Eternal—some components must be started on system power-on and run until system shutdown. Examples of these are Components involved in logging, database access, and even the Component Manager and Master applications.
- Long-lived—these components, once started, continue to operate until explicitly shutdown. Most Components are long-lived.
- Transient—other components exist only long enough to satisfy some request. An example of this might be a handler to respond to a specific error condition.

In all Components, critical state information (including initial configuration parameters and checkpoint state) is maintained in a persistent store that is maintained and accessible from outside the Component. The only exceptions are state information that varies too rapidly to be maintained externally and information that is relevant only within a single instance of a Component. Some information, such as the current Component health and the Component's life cycle model, is required to be kept in the persistent store.

A Component can be directed to reset its internal state to match one held in the persistent store and to save its current state into the persistent store (though not all Components will honor such a direction).

The OCS defines a life cycle interface that must be implemented by all Components to provide life cycle management. Details of this interface are part of the technical design and TBD.

3.2.4 Resource management

The ATST has a finite set of resources that must be shared when operating in a multiple experiment mode. Some resources are physical objects, such as detectors, filters, and CPUs. Others are less obvious, such as data stream bandwidth, target position, the light-beam itself and even time-of-day. Some resources are sharable, often up to a possibly dynamic limit (the light-beam, for example, may be shared by multiple experiments, but only if certain image quality constraints can be met) and under some conditions. Resource allocation is determined by availability, priority, and policy. When an application needs a resource it is the responsibility of the OCS to permit or deny that request. The OCS Resource Manager is responsible for making this determination.

The OCS Resource Manager operates at a coarse level and allocates resources among the major systems. Each major system also includes a Resource Manager for allocating its available resources among its

subsystems. While there is no limit imposed on the depth of this hierarchical structure, in practice there are few Resource Managers implemented below the major system level.

3.2.5 System monitoring

The OCS is responsible for monitoring and archiving system status. The system supports both polling and event posting, with the majority of status information coming from events. Status information is time and source stamped. The archiving system must support TBD status events per second continually, and TBD events per second for short (< TBD minutes) periods. The archive is searchable and clients exist for reporting search results in a meaningful fashion. Status events are monitored for alarm conditions. Actions can be assigned to be triggered on specific alarm conditions.

It is important to note that the OCS system monitoring is not responsible for ensuring system safety, but can report unsafe conditions. System safety is entirely the responsibility of the Global Interlock System.

3.3 TELESCOPE CONTROL SYSTEM

The TCS is a traditional telescope control design involving a pointing engine, a set of virtual telescopes, and various sub-systems to offload specific functionality. The TCS will meet the requirements to (a) accurately and repeatedly point to and track on the Sun, (b) direct the optical image through the appropriate path and optical elements, (c) deliver a high quality image to ATST instruments, and (d) interface to the Global Interlock System. The TCS supervises the mount, enclosure, M1, M2, feed optics and AO control systems to achieve these requirements.

3.3.1 Pointing

The TCS contains a pointing engine used in most major telescopes developed by Pat Wallace at DRAL. This engine uses several virtual telescopes to describe the pointing model for the actual telescope. Unusual features in the ATST are:

- The inclusion of the Coudé Rotator axis in the pointing model.
- The use of ecliptic and heliocentric coordinate systems.
- The lack of classical point-source guiders.
- The possible use of limb guiders outside delivered ATST light beam.
- The extensive use of open-loop tracking.
- The difficulty of assigning absolute reference positions while closed-loop tracking on solar features using AO.
- The difficulty of building pointing maps with a solar telescope.

Offsetting these points are the relaxed pointing and tracking requirements for solar observations. Few solar observations require pointing accuracies below 2 arc-seconds and offsetting abilities better than 0.1 arc-second. Many observation positions are configured manually without regard to the actual position on the sky. Positions in heliocentric coordinates are often more desirable, but often difficult to determine.

3.3.2 Trajectories

The TCS is responsible for calculating pointing and tracking trajectories for the mount and secondary sub-systems. Trajectories describe an arc these systems must follow using their servo systems. The trajectory consists of a 20 Hz stream of time stamped coordinates in the sub-systems local coordinate space.

3.3.3 Image Delivery

Pointing and tracking the telescope on solar phenomena are not sufficient. The TCS must also deliver the light to the appropriate instrument(s). The Feed Optics Control System (FOCS) operates the mirrors and other optical elements (except the AO system) past the secondary mirror. The delivered beam may go to the Gregorian station or to one of the coudé rotator stations. Any active optics wavefront sensors or pickoff mirrors are also controlled by the FOCS.

3.3.4 Image Quality

Image quality is quantified by the wavefront sensors within the Adaptive Optics system. These data are broadcast by the AO control system (AOCS) and may be received by the control systems running the AO, primary, and secondary. The TCS is responsible for coordinating these control systems by assigning specific image corrections, rates, and blending to them. The image quality data will likely be a stream of Zernike coefficients. The M1CS may use a few of the lower terms to modify the figure of the primary, while the secondary may use the tip-tilt-focus (TTF) values to correct the pointing of its hexapod. The TCS will assign the appropriate coefficients and integrations to the sub-systems.

3.4 INSTRUMENTS CONTROL SYSTEM

The ICS is responsible for managing instrument sets. An instrument set is a collection of instruments assigned to an experiment. The ICS allocates those (and other) resources to an experiment and provides a sequencer for synchronizing the operation of the instruments.. The sequencer is a management controller that can execute a script to choreograph the actions of the instruments in the instrument set.

The ICS implementation includes a template for custom control components and standard scripts for scriptable control components.

Instrument sets have the same life cycle characteristics of other applications. Facility instruments are often long-lived and exist from one experiment to the next. Other instruments may be transient and released at the end of an experiment.

3.5 DATA HANDLING SYSTEM

The DHS provides the data streams for scientific data. The major design driver is the anticipated performance requirements. Because these rates exceed current network and CPU backplane performance, detector Controllers must be capable of rudimentary data reduction operations (binning, co-adding, etc) to reduce data stream rates to acceptable levels prior to leaving the instrument.

3.5.1 Camera lines

Each camera is attached to a separate camera line within the DHS. Camera lines isolate the flow, processing, and storage for individual cameras from each other. This provides a scalable design that can adapt quickly to changes in both the number and performance characteristics of camera systems.

3.5.2 Quick look

The ATST system provides a uniform quick-look capability to aid in the dynamic assessment of image quality. Detector Components that support quick look provide suitable image-streams as low-latency data through the quick-look service. *Suitable* implies that some data reduction may need to be made to render the image-stream useful for quick look purposes.) The DHS can provide CPUs along the quick-look data stream to support this processing, but detector Component designers are responsible for providing the specific code (meeting DHS interface requirements).

Quick look client applications can subscribe to specific image streams. Initially, most quick look clients merely display the data with a few simple controls (color map selection, image scaling, image cuts, etc.). Later clients may be added for automatic quality assessment and control feedback.

3.5.3 Data store

The DHS is responsible for transferring science data to removable media (tape, removable disks, etc.).

As of the conceptual design review, there is no requirement that ATST provide a long-term archive of scientific data. Nevertheless, some temporary store is needed to collect data before transfer to removable media because of the potential transfer rate differences between data collection and removable media access. For this reason, data collection is a two step process:

- Transfer data from detector controllers to temporary storage using the reliable bulk data transport
- Transfer data from temporary storage to removable media.

The choices for removable media are both TBD and likely to change over time.

The size of temporary storage depends on the requirements for each camera line.

3.6 COMMON SERVICES FRAMEWORK

The Common Services Framework is a tiered architecture as shown in figure 3.

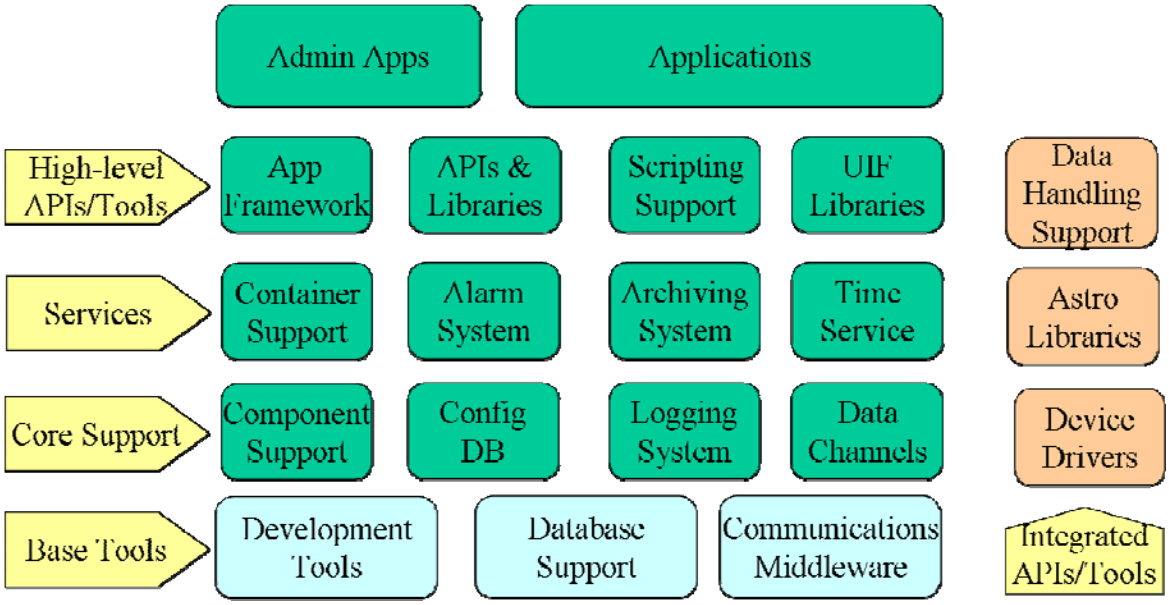

Figure 2 - Common services framework

3.6.1 High-level APIs and Tools

This level of software directly supports the development of ATST applications and is the level presenting the Common Services Framework functionality to application developers.

3.6.2 Services

The services layer consists of software that implement mid-level ATST functionality based on the core tools.

3.6.3 Core tools

The Core tools build low-level ATST required functionality directly from the base tools. Core tools often have performance constraints that mandate direct access to the foundation software.

3.6.4 Base tools

The lowest level contains software that is independent from the actual ATST functionality. It is support software on which ATST aware software is based. Most of the software at the base level is COTS or open source and is not maintained by ATST.

- Development tools include IDEs, languages and compilers, versioning systems, debuggers, profilers, and documentation generators.
- Database support includes software on which one can implement persistent stores, engineering archives, and science header and data repositories.
- The communications middleware is the foundation for all interprocess communication in ATST—it provides the communications bus, location utilities (name services), and a robust notification system.

4. TECHNICAL DESIGN

ATST is based on a distributed architecture where components are distributed across multiple computers. These computers are connected using Ethernet as the primary network layer. (Other communication networks may be used for special purpose tasks.) As much as possible these computers are treated as interchangeable and, unless dictated by hardware requirements, no specific task is required to run on a specific computer.

A fundamental precept of this architecture is that all software components are individually accessible from any other component unless access is explicitly restricted. The extent to which this accessibility is organized (hierarchically or otherwise) and how access is restricted are properties of the system design, not the infrastructure.

One goal of this baseline architecture is to isolate hardware specific operations within small components; thus increasing system flexibility. For this reason the architecture distinguishes between software devices and device drivers. Software devices are the smallest addressable components in the system. Device drivers are software components whose operating environments are physically constrained in some manner.

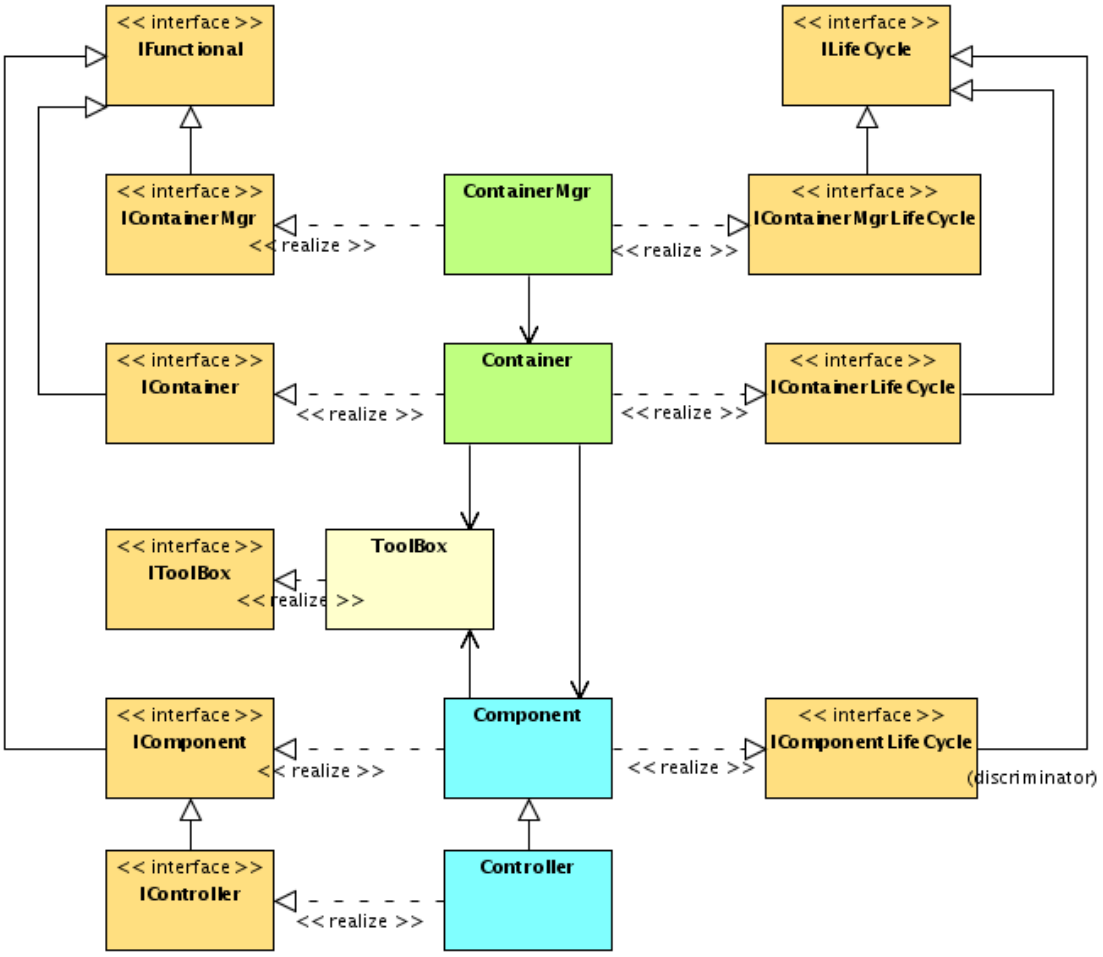

Figure 3- Component and Controller class diagrams

While the functional design focuses on providing a design that satisfies the ATST software requirements, the technical design concentrates on implementation issues.

4.1 CONTAINERS

Containers provide a common environment for managing application components. All components run within a container. A single container can *manage* multiple components and provides access to all common services. Containers provide service access by creating a *toolbox* of services for each component. Each toolbox can be configured individually with /service helpers/ that act as the interfaces between a component and the various services. Each container includes a /component manager/ that communicates with components through the lifecycle interfaces.

Containers themselves are managed by *container managers* that communicates with the various containers through the lifecycle interfaces.

Figure 4 shows an overview of the class structure for the ATST Container/Component model.

4.1.1 Components

Components are the basic building block in the control system's distributed architecture and contain the following features. The concept of a Component is used in all the following sections as an architectural foundation. Controllers are Components that follow the command-action-response model.

4.1.2 Features

Entities are identified within the control system using a three tier naming scheme:

- Component Name - a Component represents or a logical entity in the control system that is addressable through the control system infrastructure.
- Component Property - (also known as *property*) every Component contains a set of properties that are monitored and/or controlled. Properties have an associated type and may be either read-only or read/write. Read access to a property also implies access to that type's quality, which indicates the reliability of the accessed property's value.
- Property characteristic - this is static data associated with a property and includes meta-data such as type, dimensions, and description as well as other data such as units and limits.

Components can be composed hierarchically into logically organized system components. Tools are provided to navigate through this hierarchy.

This logical hierarchy is dynamic. A specific Component may be relocated within this hierarchy as part of a system reconfiguration. Unlike conventional night-time and radio observatories, where reconfiguration is an uncommon event, system reconfiguration is anticipated to be a frequent operation within ATST's laboratory environment.

All Components are named uniquely within the control system.

Component properties are maintained in a run time database that is independent of any processes used to generate and update those properties. This separates property access from Component operation as well as allowing access to properties without relying on the availability of specific server applications. It also enables the persistence of property values across system crash and reboot cycles and provides support for post-mortem debugging. Performance of this run time database is an issue affecting the performance of real time Components.

Configuration parameters are initial values for Component properties as well as property characteristics. Configuration parameters are stored in a persistent configuration database.

Components may be stateful. In particular, low level Components are all stateful and have standard state machine and transition diagrams defined for them. Some Components may have additional sub-state diagrams and transitions.

4.1.3 Implementation considerations

Components are a common way to represent control systems.

Both ACS and NDDS provide various implementations useful for representing and processing Components. ATST has added a layer on top of ICE to provide similar functionality.

Components are Java/C++ objects implementing the appropriate CSF interfaces.

Any Component hierarchy will be imposed upon a physically flat Component organization. Components themselves are unaware of any such hierarchy.

Naming services, supported by tools that implement a hierarchical logical structure, are used to impose, and provide for navigation through, a hierarchically organized logical structure. Implementing a dynamic logical hierarchy can be easily done using a wrapper around a conventional naming service paired with a persistent store.

All Components are reachable through this hierarchical structure.

The configuration database may be distributed but all nodes comprising the configuration database are synchronized periodically with a central repository. This central repository is maintained under configuration control. In the initial development, only the central repository is expected to be implemented.

4.2 PERSISTENT STORE

4.2.1 Features

The persistent store is composed of one or more databases. The databases are transaction-based, high-performance databases. Database queries are available via SQL, as are database insertions. Database accesses however, are hidden behind a persistent store interface to allow replacement of one database implementation with another. This persistent store interface provides a mapping between architectural concepts in ATST and the database model.

4.2.2 Implementation considerations

At the current time, there are a number of COTS RDBMS available that are suitable for use with ATST. ATST is using PostgreSQL.

4.3 CONTROLLERS

A Controller is a subclass of a Component, used to manipulate configurations. The Controller class is only one possible way of manipulating components, others may be implemented during the course of ATST development. However, a controller is ideally suited for many situations, especially those that need to handle multiple, simultaneous configurations.

A component does nothing with configurations, it simply manages its own lifecycle and accepts low-level get and set operations on attribute tables. Since a configuration is more than a grouping of attribute-value pairs, there needs to be a class that controls configuration lifecycle issues. Hence, the Controller class.

4.3.1 Command-action-response

The controller implements a command-action-response model. In this model, commands are submitted to the controller where they are either accepted or rejected based upon the validity of the argument and the state of the controller. If a command is accepted by the controller it causes an independent action to begin.

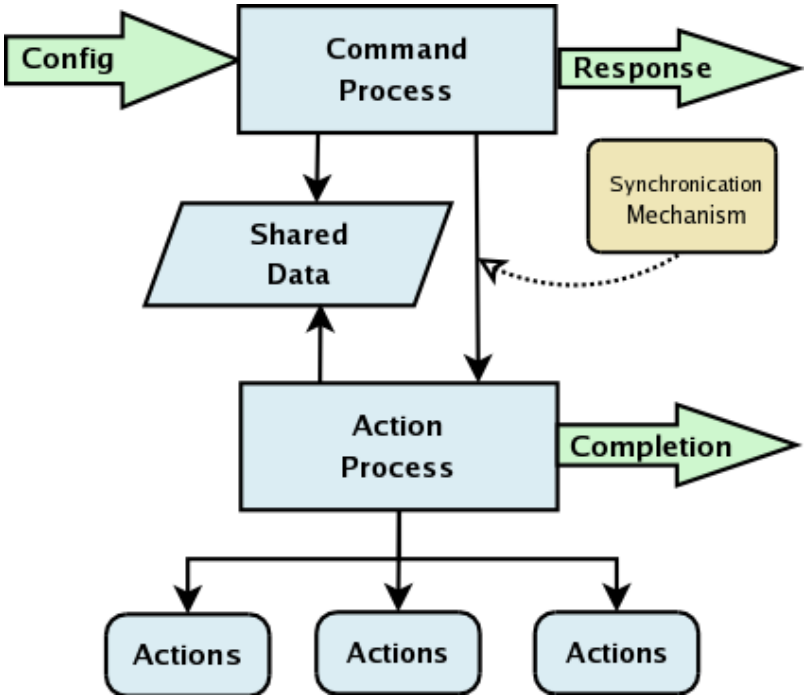

Figure 4- Command/Action/Reponse model

A response to the command is returned immediately. The action begins matching the current configuration to the new demand configuration. When the configurations match (i.e., the controller has performed the input operations) the action signals the successful end of the action. If the configurations cannot be matched (whether by hardware failure, external cancel command, timeout, or some other fault) the action signals the unsuccessful end of the action.

The important features of the command/action/response model are:

- Commands are never blocked. As soon as one command is submitted, another one can be issued. The behavior of the controller when two or more configurations are submitted can be configured on a per-controller basis.
- The actions are performed using one or more separate threads. They can be tuned for priority, number of simultaneous actions, critical resources, or any other parameters.
- Action completions produce events that tell the state of the current configuration. Actions push the lifecycle of the configuration through to completion.
- Responses may be monitored by any other Components/Controllers.

4.4 DEVICE DRIVERS

4.4.1 Features

The ATST software infrastructure assumes a Unix-like operating system throughout the system, with the possible exception of operator console systems. It is highly desirable that the choices for operating systems be standardized and that all computers be kept as similar as feasible, both in hardware and software versions.

Device drivers for standard boards are either provided as part of the operating system or as part of the control system infrastructure.

The infrastructure provides communication libraries for system communication and control across the system buses that are selected for use with ATST.

4.4.2 Implementation considerations

The SOLIS system has developed a reasonably generic model for a device and implemented that model using CORBA/C++. This provides a starting point for ATST (there are some things we have learned while working on SOLIS that would be applied to ATST). Replacing CORBA with ICE is not difficult.

4.5 MESSAGES

4.5.1 Features

A common communication protocol supports the transparent exchange of messages between system components. This exchange mechanism is independent of the location of the communication components; they may be on the same machine or running on separate hosts.

In all cases, the message system automatically selects the most efficient underlying transport mechanism for communication.

A name resolver is used, allowing one component to obtain a reference to another object by name only, without regard to that object's physical location in the system.

The architecture imposes no restrictions on the content or from of messages, though the system design may impose such restrictions on top of the architecture.

The message system supports transparent program language mapping and heterogeneous host architectures (byte swapping, etc.).

The message system supports both peer-to-peer messaging and broadcast messaging.

The message system is the foundation for all system communications, including events, alarms, and logging. As such, the message system supports the standard functionality associated with all system communication structures.

4.5.2 Implementation considerations

Both CORBA and ICE provide full support for interprocess communication with messages. CORBA C++ ORBS for VxWorks may have performance issues and SOLIS experience with CORBA indicates possible problems with overall robustness CORBA based notification services.

ICE has been selected as the baseline for the communications middleware but the ATST common services design attempts to isolate this choice as much as is reasonable to allow other options should ICE prove to be unsuitable.

4.6 CONFIGURATIONS

The ATST control system behavior is modeled as the transition of the system between states. A configuration consists of the information necessary to describe one such transition. Configurations are the fundamental objects for performing system control.

4.6.1 Features

A configuration is a set of zero or more controller properties and property values. A full configuration for a controller contains all the properties for that controller.

A controller responds to a configuration by attempting to perform the transition described by that configuration. It may not be possible to perform the transition successfully for a variety of reasons: (a) the configuration may represent an illegal transition, (b) the configuration may contain insufficient information to describe a complete transition, (c) some condition internal to the controller prevents it from performing the transition, (d) the source of the configuration is not authorized to direct the controller, etc.. All such conditions are considered error conditions.

A valid full configuration is sufficient to establish a new state for a controller regardless of its previous state.

For the most part, controllers are considered intelligent, in that they are capable of performing any required sequencing of sub-controllers needed to perform the transition defined by a configuration. However, it should be noted that external sequencing is possible through the application of a sequence of small configurations. External sequencing increases the likelihood of an error condition, however.

Configurations are also used by a controller to report some aspect of its internal state (i.e. produce the current values for a set of properties). Configurations used for this purpose are referred to as status configurations.

4.6.2 Implementation considerations

The use of configurations as sketched above has been done in the SOLIS system and adapted for use in ATST.

4.7 EVENTS

Events in the ATST control system are asynchronous messages available through a publish/subscribe mechanism.

4.7.1 Features

Events have an associated name, value, timestamp, and source. The name of the event is structured and consists of one or more event name fields.

Callbacks are set by event name, not by controller name. The event service is responsible for locating the appropriate controller, based on the event name.

The event name is always published along with the event value.

The event value may be the value of a property or it may be a status configuration.

The event timestamp is the time of posting.

A server may elect to be publish an event periodically at a given frequency or when some condition is met (most commonly when the value changes).

A client may subscribe to an event directly or may subscribe (attach a common callback) to a set of event names. In this case the callback is invoked when any event named in the set is posted.

Events posted from a single source are guaranteed to arrive at a client in the same order as posted.

4.7.2 Implementation considerations

Performance and robustness are driving factors in the implementation of ATST's event driven system.

It may become necessary to provide multiple access points to the event service, distributing the load on the event service, for performance reasons. This should only be done should it prove to be necessary, but the design should accommodate such a change easily.

A hierarchical naming system for events, with wildcards, provides a convenient method of implementing event sets. The ICE event service, IceStorm, does not support wildcarding of event names. Some form of filtering can be provided to achieve similar functionality.

4.8 LOGGING

Logging stores arbitrary status and diagnostic messages into an archive for later retrieval and analysis.

4.8.1 Features

All logged message are time-stamped and tagged with the name of the generating controller.

Messages are collected into a database on a central host and available for on-the-fly monitoring and for later analysis.

Loss of the logging system does not impede Component operation, but loss of logging information is recorded.

Control over logging is dynamic and part of the interface to every controller.

A Component may apply both categories and levels to the logging of messages.

4.8.2 Implementation considerations

Logging fits naturally on top of a notification service when coupled with a relational database.

Packages such as log4j and jdk1.4's logging interface are suitable foundations for logging, but would need to be extended into the C++ world for ATST. This could be done using an ICE interface to a logging service.

A fall back mechanism needs to be provided to handle situations where the logging service is unavailable.

For performance reasons it may be necessary to implement a logging cache service to bundle up log messages locally before relaying them to the logging service.

4.9 ERROR HANDLING

Errors are reports of the systems inability to respond to an action request.

The error system provides a standard mechanism that allows tasks to process internal errors and, if necessary, propagate errors to other tasks. Users are notified of errors that occur as a result of user initiated actions.

4.9.1 Features

Errors propagate back through the call chain, including remote method invocations.

Stack traces are included as part of the error report

The possible actions at any point of the call chain include:

- Recover from the error without logging. No trace of the error is left in the system
- Recover from the error with logging. Only the error log contains a report of the error.
- Propagate the error higher, possibly adding additional information to the error.
- Close the error, logging all the error information (including the call stack)

There are operator interfaces for monitoring errors and for examining logged errors.

All possible error conditions are predefined in error configuration files that form the basis for documentation on that error. Recovery procedures are included in these files.

4.9.2 Implementation considerations

There are two basic mechanisms for propagating errors: exceptions and completion codes. Exceptions in Java provide a convenient means of propagating errors while adding information to the error. Completion codes are more problematic. To implement the above features, completion codes would have to be more than simple integers, but instead would have to be objects that include stack traces. The same Error object should be used both exception handling of error propagation and completion code propagation.

In the case of Java, the Error class would be a simple subclass of the Exception class.

4.10 ALARMS

Alarms are reports of component failure (software or hardware) within the system. Alarms differ from errors in that Alarms can arise asynchronously.

4.10.1 Features

Alarms are defined based on condition rules for controller properties.

Alarms are reported to a central alarm handler that logs all alarms and tracks the state of all alarms.

Some alarms require explicit acknowledgment from an operator, others may simply disappear when the condition causing the alarm is cleared.

Alarms may be organized hierarchically, especially for display to an operator.

It is possible to define actions that can be automatically started on the occurrence of specific alarms.

Applications may request to be notified of alarms.

4.10.2 Implementation considerations

There is no ICE definition for a standard alarm system.

Alarms can be implemented on top of the ICE event service using callbacks, however. Alarms are logged into a RDBMS.

4.11 TIME

4.11.1 Features

The time system provides a standard abstract API to access time information in ISO format. This API is uniform across all systems, but provides higher resolution and performance on systems that have specific hardware support.

There is support for timers and the periodic triggering of events.

When time needs to be communicated between systems it is always provided as an absolute time.

There may be a need to support different types of time to support astronomy, these are TBD.

4.11.2 Implementation considerations

ATST will likely have to provide a time bus (IRIG-B?) for systems needing high time accuracy.

NTP is the baseline for time synchronization among other systems.

The source of the time signal for the time bus must also be the source for NTP time synchronization.

4.12 SCRIPTS

The ability to express tasks using a scripting language is part of the control system infrastructure.

4.12.1 Features

The scripting language provides access to the basic features found in the infrastructure, including interprocess communication facilities and database access.

Support packages written in the scripting language itself are provided as part of the infrastructure.

4.12.2 Implementation considerations

An existing, standard scripting language (or set of such languages) should be used. Possible candidates include tcl/tk, python, and ruby as well as the standard Unix shells. Python and zsh are the baseline choices. JPython is used to embed scripting support within Java-based components.

4.13 GUI TOOLS

The vast majority of user interaction with the ATST control system is through GUIs.

A high quality, uniform look and feel is a key aspect of the user interfaces to the ATST control system.

4.13.1 Features

A standard set of GUI development tools is provided, including:

- an interactive GUI builder
- a standard set of widgets
- standard support libraries for the integration with the ATST control system

GUIs are platform independent as much as feasible (there may be special circumstances for specific displays, such as an acquisition systems need to display frames rapidly). GUIs developed on any system should run on any other system.

GUIs provide access to on-line help and documentation through a browser interface.

GUI applications implement no control logic. Any action performed through a GUI can also be performed through the scripting language and other programming languages using a CLI.

4.13.2 Implementation considerations

Java and LabVIEW are the languages under consideration for use in GUI applications.

An ICE interface to LabVIEW would have to be found or implemented.

4.14 HIGH-LEVEL APPLICATION FRAMEWORK

The software infrastructure must support application development by providing a framework permitting easy development of applications in a uniform manner. Code reuse, documentation, and other aspects of software development are addressed by this framework.

4.14.1 Features

There is a base, standard model for Components.

There are standard packages supporting communications, logging, error reporting, etc.

There are code templates for typical Components and other applications (system monitors, etc.).

There are standard, base interfaces for core services and Components that are extended as needed for special applications.

There is support for automatic generation of documentation.

There is support for regression testing and debugging.

4.14.2 Implementation considerations

Example applications illustrating the use of this framework should be developed, tested and documented.

Some policies will have to be put in place to act as guides for developers.

4.15 SYSTEM CONFIGURATION AND MANAGEMENT

Building, installing, and maintaining the ATST control software is also addressed by the core infrastructure. These are major concerns in the ATST control system.

4.15.1 Features

Building of the entire system as well as building parts of the system are supported by a build tool.

A software version control repository is used to maintain a history of software changes and to identify specific system releases.

Configuration data is also maintained in a version control repository.

Documentation is also maintained in a version control repository.

The current state of the system is always available through database access.

The major system components above the infrastructure layer are as decoupled as possible. That is, modifying one system component should not require the rebuilding of other system components.

The entire control system can be started/stopped from a central site with easy to use commands.

During operation the system is monitored by one or more management applications that are capable of detecting terminated processes and (at some point) potentially restart them automatically. Similarly, resource allocation problems (lack of disk space, etc.) are also detected by management applications.

4.15.2 Implementation considerations

CVS is the repository for software. Source code documentation is maintained through *doxygen*.

5. GENERAL ISSUES

This section covers issues that need to be addressed during system development. In some sense these issues describe the philosophy of the design.

Languages will primarily be Java for high-end software and C++(g++) for software with tight performance or space considerations. Multiple scripting languages are supported, with Python the preferred choice. Zsh is used for shell programming.

In terms of databases, currently a RDBMS is expected to be sufficient. The baseline choice is PostgreSQL.

The choice of communications middleware is an important choice it must be robust, efficient, maintainable, and have clearly defined interfaces to ease building ATST services on top. ATST is currently evaluating two choices: ICE and DDS. ALMA's ICE has the advantage of having been used for ATST prototyping and is an established product. DDS has blazing performance characteristics, high-availability, and simple interfaces, but open-source versions that provide sufficient performance have only recently become available.

5.1 PORTABILITY

The higher level ATST software should be, as much as possible, independent of specific hardware and software platforms. A common approach these days is to use Java, which pushes most of the portability issues down into the Java Virtual Machine (and thus pushing most of the problems associated with maintaining portability onto the JVM developers). This is not a perfect solution, however. There are still differences in JVMs, especially has provided by Web browsers. For this reason, most Java program should run as independent applications and ATST should target support to only a few platforms.

GUI development is a major aspect of the ATST control software development. GUIs can be developed in Java and are highly portable.

5.2 MODULARITY AND DECOUPLING

Basic services, such as logging and messaging, should be independent from the other packages so they can be used in other parts of the ATST system without requiring the implementation of other control system components. Access to control system data should be similarly decoupled.

Third-party software should also be decoupled as much as feasible from the rest of the control system software by wrapping those services in generic interfaces. This helps reduce the likelihood of vendor lock and also allows extension of these services in a consistent manner.

5.3 FILE HANDLING AND LIBRARIES

The core infrastructure should provide class libraries for handling standard file format handling and other commonly needed operations. A list of these libraries needs to be compiled and standard, off-the-shelf, libraries should be used where possible. Wrapping should be employed as needed to provide a consistent interface with the rest of the system.