NGAO Software Architecture
KAON 674: Sequencer Architecture Design

Author	Modified	Notes
Douglas Morrison	7/31/09	NGAO Sequencer Architecture Design

1.0 Introduction

Sufficiently complex system built with the Keck Common Services Framework (KCSF), such as the Next Generation Adaptive Optics (NGAO), will comprise dozens or hundreds of devices and systems. During normal operation these components will need to be coordinated and managed throughout the distributed control system to perform their required tasks. Users and developers of KCSF systems will need a means of efficiently building and coordinating the command logic between these devices in an organized and repeatable way. The solution will need to manage many different types of concurrently executing commands, provide common mechanisms for control and synchronization, and be able to handle many disparate low-level interfaces.

This document details the KCSF Sequencer API and how it can be used to effectively coordinate and manage a distributed system to perform complex tasks.

2.0 Sequencer Overview

Sequencers are implemented as a state driven KCSF controller with added functionality for command execution and management. As a controller, Sequencers are capable of receiving commands, sending responses, pushing events, and triggering alarms. As with other types of KCSF components, a Sequencer instance is defined by a unique name within the system. Sequencers will implement the standard get / set / execute Controller interface, which will be used to command the sequencer and issue state transitions.

An additional benefit to developing sequencers as Controllers is that tasks can be used to allow one sequencer to command another. In this way, a hierarchy of sequencers can be created allowing the developer to organize sequencers within domains, all of which can be commanded by a single high-level multi-system command sequencer.

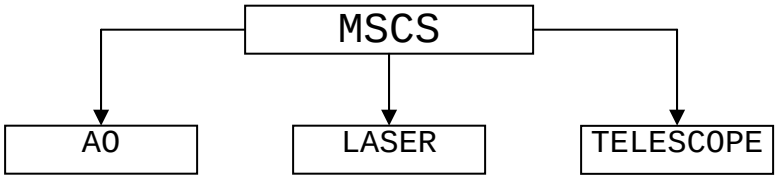

Figure 1: Multi-System Command Sequencer

States are used to organize a set of related tasks to perform a single coordinated control sequence (for example, acquiring a target). Operators issue transitions to Sequencer instances through the KCSF middleware as they would execute actions on standard device controllers. The developer is responsible for defining the states and transitions for a sequencer, and assigning the tasks that will be executed within each state.

The Sequencer's main system control functionality comes from task implementations defined in the Task Library. Sequencers are responsible for providing the execution environment for the tasks they will use. Typically a sequencer will utilize an existing lifecycle management object to perform task scheduling and execution. Alternatively a custom executor can be created to provide unique task management if required by the sequencer design. As most sequencers rely on immediate execution of tasks (as opposed to scheduled), many of which contain parallel command processing, the Task Library Command Thread-pool Executor is an ideal management mechanism for sequencers. This executor implements a queuing thread-pool, which will allow the sequencer to issue multiple commands simultaneously, as well as utilize concurrent tasks defined in the Library.

Although a task's functional requirements tend to be static after development, it is not unusual for a sequencer's requirements and control flow to evolve as the system matures, and operators' understanding of it improves. As such, the Sequencer design offers developers and users with the ability to modify the execution of a sequencer without making direct modifications to the sequencer code (this can even be done dynamically at run time). The ability to modify a sequencer's task control is provided through the KCSF Script Engine. This utility enables the loading and execution of external scripts which can be written in a number of different programming languages. Developer's can substitute state task(s) with scripting, allowing users to modify the execution logic as needed.

The following diagram details the basic Sequencer design concept and relationships.

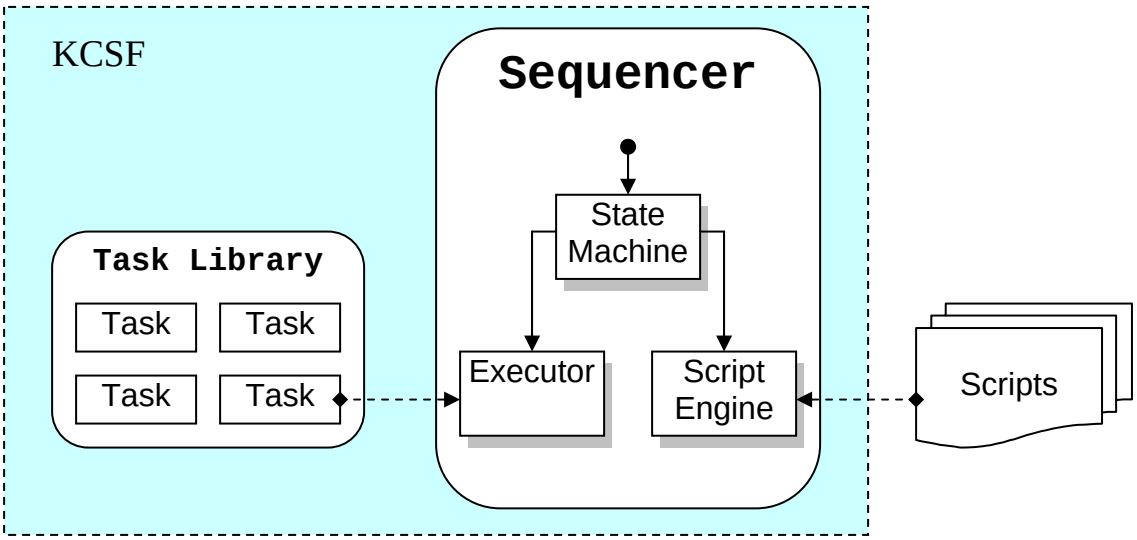

Figure 2: Sequencer Design

3.0 State Machines

Typically, Sequencers will implement a well defined State Machine to control the execution of steps in an observing sequence. The State Machine design pattern does this through the use of *states* and *transitions*. *States* represents the current configuration of the sequencer and *transitions* represent the valid paths that can be taken to a new configuration. In a state diagram, states are usually depicted as circles and transitions as arrows, both of which have an associated name.

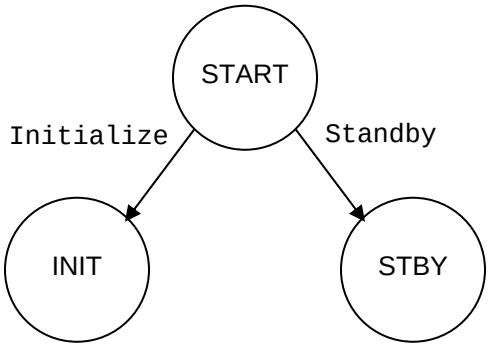

Figure 3: State Machine Diagram

The various states and transitions a sequencer will implement must be determined during the design phase of the sequencer, and is based on the intended functionality of the sequencer. It is within the transition process that the business logic of the sequencer is executed and tasks are performed.

3.1 Defining a State-Machine

Each sequencer will have its own unique state mappings based on the role it will fill. These mappings are typically built into the code of the Sequencer or represented by an external class. A suggested freeware tool that can be used to develop state mappings is the State Machine Compiler (SMC). This java application takes a file containing the user defined state-transition mappings and generates a fully operable state machine in the desired target language (e.g. Java). The Sequencer implements the business logic for the transitions, and binds to the state-machine instance. Invocating transitions on the state-machine will execute the corresponding tasks and set the new state.

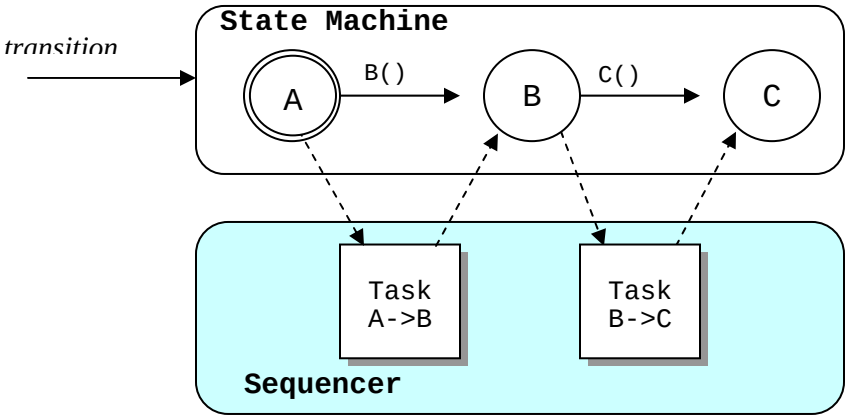

Figure 4: Executing Transition Logic

Developing a state-machine, and by extension a Sequencer, requires careful consideration to account for all of the intended responsibilities of the design. Each state should reflect a specific configuration or step identified within a science or motion control sequence.

3.1.1 Common States

Although each Sequencer will have many unique states, there are a set of states that will be common to all Sequencers. Theses states are derived from the standard state-machine design used by hardware devices – the control targets for the majority of Sequencer implementations.

- *START* – the entry state of the Sequencer. It is assumed that a Sequencer in this state has recently been created, but not yet initialized or configured.
- *INIT* – the initialization and configuration state of the Sequencer. Typically the operations performed during this state only need to be performed once after startup. This may include gathering configuration information, creating and initializing class members, and obtaining services.
- *REINIT* – a fast or repeatable initialization phase. This state typically implements a set of the Sequencer initialization that may have to be repeated multiple times during a night (e.g. connecting to devices, refreshing telemetry, etc.) Usually execution of the INIT state will automatically transition through REINIT, and then to STANDBY.
- *HALT* – indicates that an operation has been interrupted. This is usually entered by explicit command from the operator.
- *STANDBY* – the Sequencer is in an idle state, and is ready to receive and process commands.
- *SHUTDOWN* – the termination state of the Sequencer. This state is responsible for closing connections, releasing resources, and shutting down the Sequencer.
- *FAULT* – this state indicates that an unhandled or non-recoverable error has occurred and the sequencer had to stop. Operator intervention is expected to resolve the problem.

In addition, Sequencers that are responsible for acquiring targets or positioning devices will also typically use a SLEW / TRACK state pattern. In this design, any Sequencer operation that involves starting a process and then waiting for one or more devices to achieve a specific state will quickly execute the task(s), and enter a SLEW state. The sequencer will wait in the SLEW state until the tasks are complete.

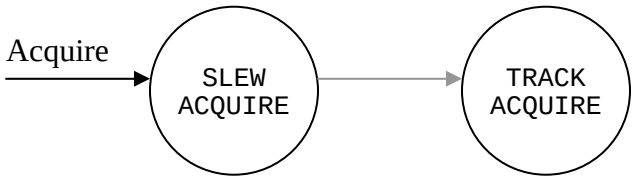

Figure 5: SLEW-TRACK States

The SLEW state is known as a *transient* state: a temporary state that will automatically transition out when an internal event occurs (i.e. not caused by an explicit user action). When the devices have reported in position the task will issue its own transition to move to the TRACK state. As with other states, if there is a problem during the task execution of a transient state the state machine will typically enter FAULT to signal a system error.

3.1.2 Developing with SMC

The State Machine Compiler library is used to convert state-transition mappings into a callback processing class. Developers define a state machine using the SMC syntax. For each state, the mapping will define each of the available transitions and the target state when the transition completes. The following example illustrates a simple SMC mapping for a single state.

```
START
   {
   // START will recognize and accept an Init(void) transition.
   Init()
         // When the transition completes the state machine will
         // be put into the INIT state.
         INIT
         {
               // The transition process will execute the
               // InitializeTask routine defined by the Sequencer.
               initializeTask();
         }

   Fault()
         FAULT
         {
         }
   }
```

The compiler will read this definition in the following way:

- The state machine will posses a START state.
- The START state can be left by issuing an *Init* or *Fault* transition.
- When an *Init* is issued the machine will transition to the INIT state, after executing *InitializeTask*.
- When a *Fault* is issued the machine will transition directly to the FAULT state.

Only those transitions and tasks defined in the mapping will be permitted. If the client attempts to issue a transition not recognized by the current state an exception will be thrown. The following pseudo-code details the interface produced when the mapping is compiled.

```
public class GeneratedStateMachine  {
   public GeneratedStateMachine(<Type> machineImplementation);

   public void Init();
   public void Fault();
}
```

The *machineImplementation* parameter of the class constructor refers to the actual object that implements the state machine transition tasks (e.g. *InitializeTask*). This will be a reference to the Sequencer itself.

```
public class Sequencer extends Controller {
   public Sequencer(…) {
         this.stateMachine = new GeneratedStateMachine(this);
   }

   public void initializeTask() {
         // Implements the actual work to be performed for an INIT
```

```
    }

    ...
}
```

The SMC syntax offers additional capabilities including transition parameters, conditions, guards, code injection, default transitions and more. See the appendix for more information on the SMC compiler.

3.2 Handling Transitions

As a Controller subclass, Sequencers will receive transition requests from clients through the *execute* method. Transitions are defined in Attribute Lists in a similar way as standard device commands,

- The reserved *_Action* keyword defines the transition. This may be a string, integer, or other compatible type.
- Parameters custom to the transition will be defined in additional attributes. It is the responsibility of the client to be aware of all the attributes required by the sequencer.

The *doExecute* of the Sequencer is responsible for transitioning the state machine, and may be implemented in the following way.

```
void doExecute(IAttributeList command) {
   int transition = command.getString("_Action");
   switch(transition) {
         case INIT:
               // Get additional parameters, if required by transition.
               // Issue transition.
               this.stateMachine.Init();
               break;
         ...
   }
}
```

When executing a transition on the internal state machine, the Sequencer's thread will block until the transition task has completed, and the state machine enters the target state. This should be kept in mind when determining the default size for the Controller's thread pool. At a minimum two threads should be active to allow the user to halt or asynchronously command the Sequencer while a transition is being processed.

4.0 Tasks and Executors

The business logic of a Sequencer is executed during state transitions when the callbacks are performed. A developer can implement the functional requirements of a transition in a number of ways, but an effective technique is to use KCSF Tasks. Tasks are an ideal solution for implementing sequencer control logic because they provide a simple, scalable, and composable method to build complex command sequences. By design, Tasks are generic and reusable, allowing developers to mix and match commands to implement any imaginable system control requirements. As new tasks are developed they can be saved to the Task library further expanding the capabilities available to developers.

As mentioned earlier, Tasks must be executed within the environment of their parent application. Although the exact implementation of a task (synchronous or asynchronous) will dictate the requirements set on the execution environment, the majority of Sequencer tasks will require parallel task processing and execution. The KCSF Task Library provides a Command Thread-pool Executor that will allow Sequencer developers to fully utilize concurrent and complex tasks defined in the library.

4.1 Using Tasks

The Task library is publicly available to all users of the Keck Common Services Framework. The Task module will contain a number of implementations from simple tasks that control a single device, to complex nested tasks that can command an entire system. Usually a task will be designed to perform a single logical function (e.g., positioning a stage or acquiring a target with the AO). Compound tasks are formed by the combination of simple tasks with sequential and concurrent meta-tasks. The Sequencer developer would use these components to build a command sequence to be executed during a state transition.

```
public class Sequencer : public Controller {
   public Sequencer() {
         ...
   }

   public void initializeTask() {
         OpenShutter shutter = new OpenShutter(“ngao.ao.prishutter”);
         AlignSensor sensor = new AlignSensor(“ngao.ao.wfs”, “ngao.ao.wl”, 2.0);
         Task [] tasks = new Task[2];
         tasks[0] = shutter;
         tasks[1] = sensor;

         SequenceTask seq = new SequenceTask(tasks);

         seq.initialize();
         seq.run();
   }

   ...
}
```

This example details the creation and execution of a simple sequential compound task. The *initializeTask* method is executed by the State Machine object during the transition to INIT. Task instances are created of the hypothetical *OpenShutter* and *AlignSensor* library tasks. OpenShutter

represents a simple task that opens a single shutter. The AlignSensor task represents a compound task that will control a sensor and light source with the desired voltage. The instances are added to an array, and feed to the constructor of a sequential task. The sequential task is then initialized and run. Once the task completes the transition callback will return and the state machine will move to the INIT state.

Since tasks are fully self contained they can be created at an early stage in the life cycle of the Sequencer and simply invoked during the transition callbacks. For example, all of the tasks created in initializeTask, could be created in the constructor, saved as a class member, and executed during the transition.

```
public class Sequencer : public Controller {
   public Sequencer() {
         OpenShutter shutter = new OpenShutter(“ngao.ao.prishutter”);
         AlignSensor sensor = new AlignSensor(“ngao.ao.wfs”, “ngao.ao.wl”, 2.0);
         Task [] tasks = new Task[2];
         tasks[0] = shutter;
         tasks[1] = sensor;

         this.InitializeTask = new SequenceTask(tasks);
   }

   public void initializeTask() {
         this.InitializeTask.initialize();
         this.InitializeTask.run();
   }

   ...
}
```

4.2 Selecting an Executor

The Command Thread-pool Executor is designed to provide an application with queued management and multi-thread execution of tasks.

T.B.D

5.0 Scripting

The Framework provides sequencer developers and users with the flexibility to modify an observing sequence or its functionality without impacting the sequencer code itself. This is done through the KCSF Script Engine. This utility allows developers to load and process external scripts used to control the execution of state tasks dynamically (even at runtime).

The business logic of a sequencer exists in the implementation of the state machine transition callbacks. Up till now we have discussed executing Tasks during these callbacks to perform the system control. However an alternate solution (or to be used in conjunction with Tasks) is to have the callback load and execute script(s) defined by the developer, and specified in configuration.

```
public void initializeTask() {
   script = ScriptEngine.load(this.pathToInitScript);
   script.execute();
}
```

The script would be responsible for providing the functionality for the transitions. Since scripts executed through the Script Engine are able to use the KCSF services and can be given access to class members, a script developer can utilize any of the tasks and functionality available to the sequencer itself. Operators can fine tune script properties as the sequencer is running, or can modify the entire sequence if a better solution is discovered.

6.0 Deployment

As a KCSF controller subclass, Sequencer instances are deployed using Containers. As with other deployable components, it is up to the developer / operator to properly balance the load of each container host to make sure there are sufficient resources available for use. Multiple sequencers and components can be deployed per container, or a dedicated container can be created for each Sequencer. (The latter is the safest since you would be able to restart an individual sequencer without affecting others.)

When a Sequencer is deployed by a Container it will be initialized and put it into its command-ready state. At this point clients can connect to the Sequencer and issue transitions.

7.0 Commanding a Sequencer

Clients can connect to a sequencer instance by obtaining a proxy reference from the Connection Service. It will be the responsibility of the sequencer's parent Container to perform the initialization of the sequencer to prepare it for commanding. Once the client has a valid proxy they can issue transitions to the sequencer.

Transitions are issued through the sequencer's *execute* method by configuring a CommandSet with the required transition information. Each transition may have its own unique parameters so clients must be aware of what each transition requires.

```
CommandSet command = new CommandSet("Acquire");
command.set("TargetName", "HD13089");

if(!TelescopeSequencer.execute(command, null)) {
   // Failed to issue command. Reconnect to Sequencer?
}
```

In this example a transition request is created to acquire a target with the telescope sequencer. For this transition only one argument is required, *TargetName*. This argument is used to lookup target information (such as coordinates) in a star catalog. The command is then sent to the sequencer through the execute method.

The advantage of using the Controller *execute* method to dispatch transitions is that it will utilize the controller thread pool to allow the Sequencer to operate asynchronously. This permits a user to interrupt or alter a sequence by issuing a *halt*, *standby*, etc.

References

State Machine Compiler (SMC) Website:
http://smc.sourceforge.net/

KCSF Script Engine

KCSF Tasks