Problem

How does ACS support writing Subsystem Master Components?

Solution

What is a Subsystem Master Component?

An ACS component that represents an ALMA subsystem ('subsystem' in its technical meaning) toward the rest of the ALMA software system. It manages lifecycle details and provides information on the current subsystem state and a number of modes.

The concept was discussed at LeadsMeetingNov03, and was captured in SubsystemStateModel. The main features are the extensible models for nested subsystem states and orthogonal subsystem modes.

Using it

The master component interfaces are defined in `mastercomp_if.idl` (CVS module `ACS/LGPL/CommonSoftware/mastercomp`, or simply to be found under `$ACSROOT/idl`). There are two categories of users:

Executive (and subsystem tests)

The Executive will use all master components to coordinate the start and shutdown of the ALMA system, as well as to deliver information about the running system. It will use the full-fledged `ZLegacy/ACS.MasterComponent` interface.
To test the behavior of your master component, you are likely to also use the `ZLegacy/ACS.MasterComponent` interface from a TAT script or unit test.

Other subsystems

Any subsystem that's interested in monitoring changes in state or mode of another subsystem can subscribe to that system's master component. It must use the read-only `MasterComponentReadOnly` interface, which is the base interface of the full `ZLegacy/ACS.MasterComponent`. Here is a Java example of how to access the read-only interface:
```
org.omg.CORBA.Object compObj = getContainerServices().getComponent("MASTERCOMP_SCHEDULER");
MasterComponentReadOnly maComp = alma.ACS.MasterComponentReadOnlyHelper.narrow(compObj);.
```

The current state is implemented as the BACI `ROstringSeq` property `currentStateHierarchy`, which can be retrieved and monitored. It returns a snapshot of the current state hierarchy as a sequence of strings, with composite state names listed before their substates. For example, during start-up, the master component will temporarily be in the state that is specified by the three strings AVAILABLE,OFFLINE,INITIALIZING_PASS1.

How should a Master Component be implemented?

ACS release 3.1 introduces support for a Java implementation of a subsystem master component. It is strongly recommended to build all master components based on this; however, it would also be possible to implement the given IDL interface differently from scratch, as long as all semantics (especially the state machine) are implemented and maintained correctly.

ACS provides the abstract component implementation class `alma.ACS.MasterComponentImpl.MasterComponentImplBase` (inside `mastercomp.jar`) which implements the required event handling.

Concrete master component classes should be implemented as subclasses. They must provide an implementation of the interface `alma.ACS.MasterComponentImpl.statemachine.AlmaSubsystemActions`, which is the set of action methods invoked by the state machine in response to received events. The diagram of all actions together with the events and activity states is here.

- `initSubsysPass1`: create subsystem components and channels which don't require resources from outside your subsystem; the Executive will call this at startup for all subsystems before moving on to calling
- `initSubsysPass2`: create subsystem components and channels which do require resources from other subsystems; the 2-pass initialization is intended to resolve mutual dependencies among subsystems, with the Executive knowing the correct sequence.
- other actions are similar; possibly more actions will be added in the future.
- any of your action methods may throw an `AcsStateActionException`; this is recognized by the state machine, and will trigger a transition to the error state.

As described in the ACS 6.0 release notes, the master component base class also offers methods that allow to easily monitor the subsystem's components or other resources. The master component can be notified of and react to problems, at three different levels of detailed control vs. ease of programming.

Are there any examples?

Simple Example

For a straightforward example, see `alma.ACS.MasterComponentImpl.TestMasterComponentImpl` in `ACS/LGPL/CommonSoftware/mastercomp/test`:
`public class TestMasterComponentImpl extends MasterComponentImplBase implements AlmaSubsystemActions`,
which implements the action interface directly and returns it in the required method `getActionHandler()`.

Fancier Example

Another example shows

1. how to extend the standard master component's IDL interface for subsystem-internal communication, and still use `MasterComponentImplBase` as the implementation base class.
2. how to add other baci properties to the IDL interface (see below about CDB issues)

From mastercompTest_if.idl:

```
interface SpecialTestMasterComponent : MasterComponent
{
  void componentNeedsAttention(in string componentName, in string troubleCode);
  readonly attribute ACS::RWdouble someOtherProperty;
};
```

In this example, all subsystem components could notify their master component by calling `componentNeedsAttention(..)` when they are in trouble and feel that the master component should re-evaluate the situation and perhaps take counter-measures.
The implementation can be found in CVS: mastercomp/test, package `alma.ACS.SpecialTestMasterComponentImpl`.

CDB requirements

Because a master component extends BACI characteristic component (in order to add a property which is used to maintain the state of the component), it is necessary to configure your CDB for this BACI property. So, in addition to the "normal" CDB configuration in `/some-path-to-your/CDB/MACI/Components/Components.xml`, you will also need `/some-path-to-your/CDB/alma/YOURMASTERCOMPONENTNAME/YOURMASTERCOMPONENTNAME.xml`, which will define the baci property, etc. For an example, see `ACS/LGPL/CommonSoftware/mastercomp/test/CDB/alma/MASTERCOMP1/MASTERCOMP1.xml` and `../MASTERCOMP2/MASTERCOMP2.xml`.

Right - if you happen to forget this, you'll see something like:
Failed to activate component MY_MASTERCOMPONENT, problem was: alma.acs.component.ComponentLifecycleException: java.lang.NullPointerException: Failed to get DAO for 'alma/MY_MASTERCOMPONENT'.
-- MarcusSchilling - 05 Aug 2004

Convenience Classes for Clients

To reduce the coding effort involved in registering a listener (monitor) on a master component's `currentStateHierarchy` property, Java clients can use the class `alma.ACS.MasterComponentImpl.StateChangeListener`:

```
StateChangeListener listener = new StateChangeListener(m_logger);
listener.createMonitor(statesProperty, containerServices);
```

This will connect the `listener` to `statesProperty`, and every state change will be logged.

If you subclass `StateChangeListener` and override the method `protected void stateChangedNotification(..)`, you can implement whatever behavior is required other than logging of state changes (which is still available by calling `logNotification(..)` explicitly).

A special situation is the synchronization of your client program with state changes in the master component. Remember that activity states (those with a `/do` method in the UML diagram) are executed asynchronously. For example, a master component in state `SUBSYSSTATE_SHUTDOWN` receives the call `doTransition(SUBSYSEVENT_INITPASS1)`, changes into the activity state `SUBSYSSTATE_INITIALIZING_PASS1` and returns immediately, while the `initSubsysPass1` action may still be running. Only when this action has finished, the state machine advances to the next state, which is `SUBSYSSTATE_PREINITIALIZED`. While in `SUBSYSSTATE_INITIALIZING_PASS1`, the state machine will reject any event it receives, so a smart client will defer sending the next event until the state has changed to `SUBSYSSTATE_PREINITIALIZED`. Exactly this waiting for state changes can be achieved with the `StateChangeListener` and the associated `StateChangeSemaphore`:

```
StateChangeSemaphore sync = listener.getStateChangeSemaphore();

sync.reset();
m_masterComp.doTransition(SubsystemStateEvent.SUBSYSEVENT_INITPASS1);
sync.waitForStateChanges(2);

sync.reset();
m_masterComp.doTransition(SubsystemStateEvent.SUBSYSEVENT_INITPASS2);
sync.waitForStateChanges(2);
```

The argument of the method `waitForStateChanges` gives the number of state changes to be waited for, counting from the last call to `reset`. Here we want to block client execution until the second state change occurs, with the first one being the immediate change into the activity state, and the second one being the state change following the completion of the activity state.

What's still missing?

The current (ACS 3.1) implementation does not yet support all features, such as subsystem modes or subsystem-specific substates of AVAILABLE /ONLINE and AVAILABLE/OPERATIONAL. Also the interactions between `MasterComponentImplBase` and its subclasses should probably be extended.

However, the missing features are foreseen in the design. We hope that the necessary changes will be backward compatible. Deferring these features is also intended to first gain experience and collect feedback, which the next phase will benefit from.

There are already discussions going on (SubsystemStateModel).

Related articles

- How can more people do development with ACS on the same machine without disturbing each other?
- Which ports are used by ACS?
- Problems connecting to ACS servers on a remote machine: bad /etc/hosts
- Why does the getComponent method of ZLegacy/ACS.ContainerServices return an object of type None?
- Why are some of my print statements not showing up in the container output section of acscommandcenter?