Orchestration Server Developer's Guide

Orchestration Server How-To

4/3/2025

Contents

Orchestration Server How-To

Timers and Wait Functions

In SCXML, orchestration logic can implement timers and wait functions using the <send> element and the delay attribute. A state can send a delayed event to itself and then when the time expires the event will be processed by the `<transition>` element defined in the state to process it.

Modularity

There are several ways to support modularity and reusability with SCXML:

- xinclude <xi:include /> - This is an XML standard for including other documents into another XML document, byproviding a macro-like functionality. Note: For details on how the orchestration platform will support this, see **<xi:include>** in the SCXML Technical Reference.
- <invoke> - This creates a stand-alone sub-state machine that communicates asynchronously with its parent. See SCXML <invoke> for details.
- Targetless transitions within a state can provide subroutine-like functionality for the state and all its contained states. What is needed are two events:
  - The input event - This defines the name of the subroutine and the input parameters to the subroutine.
  - The output event - This defines the event that the invoking state should wait for to get the results of the subroutine.

The subroutine is implemented as follows:

-
  - Either in `<onentry>` or within one or more child `<states>`, generate the input event. This can be done using `<raise>/<event>`.
  - Define a targetless transition in the parent state that will handle the input event. This will provide the body of the subroutine. The body of the subroutine should generate the output event. This can be done using <raise>/<event> or by calling an action that will generate the output event.
  - Define a transition that will handle the output event. This transition can access the information in the output event to determine the results of the subroutine. This transition can be targetless. If targetless, it can act as another step in a more complex subroutine. If it is not targetless, it should be defined at the same level where the subroutine's input event was generated.

The following is an example of a targetless transition style subroutine. The input event is "inputsub1" and the output event is "outputsub1". Several steps are chained together to provide a moderately complex subroutine. This mechanism should only be used with events generated by the steps of the subroutine; otherwise asynchronous events generated by actions could result in steps not being executed properly.

```
<state id="ParentState">
```

```
        <transition event="inputsub1">
                <script>
                        local.eMailID =
_genesys.getValue(_event.data.i_ixn,"'InteractionId'");
                </script>
                <log expr="_event.data.i_message" level="3"/>
                <session:fetch requestid="_data.reqid" srcexpr="'someURL/AUDIT_PROC'"
timeout="10">
                        <param name="audit_info" expr="_event.data.i_message"/>
                        <param name="message_id" expr="local.eMailID"/>
                </session:fetch>
        </transition>
                <!-- This an example of branching within a targetless transition subroutine --
>
        <!-- It examines the event generated by the session:fetch action -->
        <!-- If value1 is less than or equal to 10 go to step 2. -->
        <transition event="session.fetch.done" cond="(_event.requestid == _data.reqid) &&
(_event.data.content.value1 <= "10")">
                <raise event="sub1step2">
                        < param name ="s1v1" expr ="_event.data.value1"/>
                </raise>
        </transition>
                <!-- If value1 is greater than to 10 go to step 3. -->
        <transition event="session.fetch.done" cond="(_event.requestid == _data.reqid) &&
(_event.data.content.value1 > 10)">
                <raise event="sub1step3">
                        < param name ="s1v1" expr ="_event.data.value1"/>
                </raise >
        </transition>
        <!-- This is the processing for step 2 of the subroutine sub1 -->
        <transition event="sub1step2" >
                <script>
                        <! - do some extra processing -->
                </script>
                <!-- Return to the involving state -->
                <raise event="outputsub1">
                        <param name ="sub1op1" expr ="variablex"/>
                        <param name ="rc" expr ="success"/>
                </raise>
        </transition>
        <!-- This is the processing for step 3 of the subroutine sub1 -->
        <transition event="sub1step3" >
                <script>
                        <!-- do some extra processing -->
                </script>
                <if conn="variable >=100">
                        <!-- Return to the involving state -->
                        <raise event="outputsub1">
                                <param name ="sub1op1" expr ="variablex"/>
                                <param name ="rc" expr ="success"/>
                        </raise>
                </else >
                        <!-- go to step 4 -->
                        <raise event="sub1step4">
                                <param name ="sub1op1" expr ="variablex"/>
                                <param name ="sub1op1" expr ="variabley"/>
                        </raise>
                </if>
        </transition>
        <!-- This is the processing for step 4 of the subroutine sub1 -->
        <transition event="sub1step4" >
                <script>
                        <!-- do some extra processing -->
```

```
                </script>
                <!-- Return to the involving state -->
                <raise event="outputsub1">
                        <param name ="sub1op1" expr ="variablex"/>
                        <param name ="rc" expr ="success"/>
                </raise>
        </transition>
        <!-- General error processing for this sub1 -->
        <!-- Note that this will treat all error events as a failure of this subroutine -->
        <!-- Care should be taken to ensure that this is only called as part of the
subroutine -->
        <transition event="error.*" cond="_event.requestid == _data.reqid">
                <log expr="had an error with the fetch" level="3"/>
                <raise event="outputsub1">
                        < param name ="rc" expr ="fetchfailed"/>
                </raise >
        </transition>
        <!-- This is the state that invokes the subroutine -->
        <state id="stepwhichinvokessub1">
                <onentry>
                        <raise name ="inputsub1">
                                <param name ="i_message" expr ="'here is the message'">
                                <param name ="i_ixn" expr ="_data.interaction"">
                        </raise">
                </onentry>
                <transition event="outputsub1">
                        <if cond = (_event.data.rc == "success")>
                                <!-- do the processing to continue based on sub1 completing --
>
                        </else >
                                <!-- do the processing to continue based on sub1 failing -->
                        </if>
                </transition>
        </state>
</state>
```

Using <invoke>

In addition to what the SCXML specification defines, the following guidelines can be followed when developing an asynchronous subroutine with this functionality:

- Invoked document/session:
  - Should use `<datamodel>` and `<data>` for mapping of the invoking session's `<param>`s to the invoked session's data model. This is the primary means of passing data to the invoked session.
  - Must use the `<donedata>` element as a child of the top-level `<final>` state(s). This will allow the invoked state machine to return the appropriate output parameters in the done.invoke event. This event is sent to the invoking session.
  - Must not send explicit events to the including document/session.
  - Should not rely on events from the including document/session, however, the subroutine can use cancel.invoke to perform any cleanup necessary if the subroutine is interrupted. This can happen if the parent session transitions out of the invoking state.
- Invoking document/session:
  - Should use `<param>` to provide argument information to the invoked session. Since the sessions do

not share a `<datamodel>`, this is the primary means for passing data to the invoked session.

- Must have a `<transition>` for the done.invoke event that will be generated by the invoked session. This allows the invoked session to communicate the subroutine results back to the invoking document/session and transition to the next state based on the results.

Handling Assembly and Compilation Problems

When using xinclude, developers should ensure that valid documents and fragments are used. This will avoid many XML parsing problems. However, since large SCXML documents can be very complicated, it is still possible to have document errors that prevent the final application from being assembled, parsed, and compiled. When dealing with such a situation, a developer can place the following in the main application document:

```
<!-- $$_GENESYS_DEBUGGING_$$ -->
```

If the application document cannot be parsed and compiled, and this element is present in the document, the entire application document will be written to a file in the current working directory. The filename will be sessionid.scxml, where sessionid is the ID of the session that was being created. The information contained in this file should be sufficient to allow the developer to determine why the failure occurred. These files need to be removed manually by the developer after the problems have been resolved.