W3C Recommendation

State Chart XML (SCXML): State Machine Notation for Control Abstraction

W3C Recommendation 1 September 2015

This version:
http://www.w3.org/TR/2015/REC-scxml-20150901/
Latest version:
http://www.w3.org/TR/scxml/
Previous version:
http://www.w3.org/TR/2015/PR-scxml-20150430/
Editors:
Jim Barnett, Genesys (Editor-in-Chief)
Rahul Akolkar, IBM
RJ Auburn, Voxeo
Michael Bodell, (until 2012, when at Microsoft)
Daniel C. Burnett, Voxeo
Jerry Carter, (until 2008, when at Nuance)
Scott McGlashan, (until 2011, when at HP)
Torbjörn Lager, Invited Expert
Mark Helbing, (until 2006, when at Nuance)
Rafah Hosn, (until 2008, when at IBM)
T.V. Raman, (until 2005, when at IBM)
Klaus Reifenrath, (until 2006, when at Nuance)
No'am Rosenthal, (until 2009, when at Nokia)
Johan Roxendal, Invited Expert

Please refer to the **errata** for this document, which may include normative corrections.

See also **translations**.

Abstract

This document describes SCXML, or the "State Chart extensible Markup Language". SCXML provides a generic state-machine based execution environment based on CCXML and Harel State Tables.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This is the Recommendation of SCXML Version 1.0. It has been published by the Voice Browser Working Group, which is part of the Voice Browser Activity.

Comments are welcome at www-voice@w3.org (archive). See W3C mailing list and archive usage guidelines.

This specification has been widely reviewed (see the Third Last Call Working Draft Disposition of Comments) and satisfies the Working Group's technical requirements. A list of implementations is included in the SCXML 1.0 Implementation Report, along with the associated test suite. The Working Group removed an unused reference to ECMASCRIPT-327 based on a public comment, updated the broken link for E4X, and fixed typos. However, there are no substantial changes from the 30 April Proposded Recommendation.

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document has been produced as part of the Voice Browser Activity (activity statement), following the procedures set out for the W3C Process. The authors of this document are members of the Voice Browser Working Group.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 14 October 2005 W3C Process Document.

The sections of this document are normative unless otherwise specified.

Table of Contents

1 Terminology
2 Overview
3 Core Constructs
  3.1 Introduction

3.2 <scxml>
3.3 <state>
3.4 <parallel>
3.5 <transition>
3.6 <initial>
3.7 <final>
3.8 <onentry>
3.9 <onexit>
3.10 <history>
3.11 Legal State Configurations and Specifications
3.12 SCXML Events
3.13 Selecting and Executing Transitions
3.14 IDs
4 Executable Content
4.1 Introduction
4.2 <raise>
4.3 <if>
4.4 <elseif>
4.5 <else>
4.6 <foreach>
4.7 <log>
4.8 Other Executable Content
4.9 Evaluation of Executable Content
4.10 Extensibility of Executable Content
5 Data Model and Data Manipulation
5.1 Introduction
5.2 <datamodel>
5.3 <data>
5.4 <assign>
5.5 <donedata>
5.6 <content>
5.7 <param>
5.8 <script>
5.9 Expressions
5.10 System Variables
6 External Communications
6.1 Introduction
6.2 <send>
6.3 <cancel>
6.4 <invoke>
6.5 <finalize>

Appendices

A Conformance
A.1 Conforming Documents
A.2 Conforming Processors
B Data Models
B.1 The Null Data Model
B.2 The ECMAScript Data Model
C Event I/O Processors
C.1 SCXML Event I/O Processor
C.2 Basic HTTP Event I/O Processor
D Algorithm for SCXML Interpretation
E Schema
F Related Work
G Examples
G.1 Language Overview
G.2 Microwave Example
G.3 Microwave Example (Using parallel)
G.4 Calculator Example
G.5 Examples of Invoke and finalize
G.6 Inline Content and Namespaces
G.7 Custom Action Elements
H MIME Type
H.1 Registration of MIME media type application/scxml+xml
H.2 Fragment Identifiers
I References
I.1 Normative References
I.2 Informative References

---

1 Terminology

The key words *must*, *must not*, *required*, *shall*, *shall not*, *should*, *should not*, *recommended*, *may*, and *optional* in this specification are to be interpreted as described in [RFC 2119].

The terms base URI and relative URI are used in this specification as they are defined in [RFC 2396].

All sections not marked as "informative" are normative.

2 Overview

[This section is informative.]

This document outlines State Chart XML (SCXML), which is a general-purpose event-based state machine language that combines concepts from CCXML and Harel State Tables. CCXML [CCXML 1.0] is an event-based state machine language designed to support call control features in Voice Applications (specifically including VoiceXML but not limited to it). The CCXML 1.0 specification defines both a state machine and event handing syntax and a standardized set of call control elements. Harel State Tables are a state machine notation that was developed by the mathematician David Harel [Harel and Politi] and is included in UML [UML 2.3]. They offer a clean and well-thought out semantics for sophisticated constructs such as parallel states. They have been defined as a graphical specification language, however, and hence do not have an XML representation. The goal of this document is to combine Harel semantics with an XML syntax that is a logical extension of CCXML's state and event notation.

3 Core Constructs presents the core state machine concepts, while *4 Executable Content* contains an extensible set of actions that the state machine can take in response to events. *5 Data Model and Data Manipulation* defines constructs for storing and modifying data, while *6 External Communications* provides the capability of communicating with external entities.

3 Core Constructs

3.1 Introduction

[This section is informative.]

3.1.1 Basic State Machine Notation

The most basic state machine concepts are *3.3 <state>*, *3.5 <transition>* and event (*3.12 SCXML Events*). Each state contains a set of transitions that define how it reacts to events. Events can be generated by the state machine itself or by external entities. In a traditional state machine, the machine is always in a single state. This state is called the active state. When an event occurs, the state machine checks the transitions that are defined in the active state. If it finds one that matches the event, it moves from the active state to the state specified by the transition (called the "target" of the transition.) Thus the target state becomes the new active state.

The Harel state notation defines several extensions to these basic notions. First of all, the state machine may take actions (as defined in *4 Executable Content*) while taking transitions. Specifically, each state may contain *3.8 <onentry>* and *3.9 <onexit>* actions. Transitions may also contain actions. If a state machine takes transition T from state S1 to state S2, it first performs the onexit actions in S1, then the actions in T, then the onentry actions in S2. Secondly, in addition to the 'event' attribute that specifies the event(s) that can trigger it, transitions also have a 'cond' attribute. If a transition has both 'event' and 'cond' attributes, it will be selected only if an event is raised whose name matches the 'event' attribute (see *3.12.1 Event Descriptors* for details) and the 'cond' condition evaluates to true. If the 'event' attribute is missing, the transition is taken whenever the 'cond' evaluates to true. If more than one transition matches, the first one in document order will be taken. Thus, in the following example, the system will transition to s1 when event e (or e.foo, etc.) occurs if x is equal to 1, but will transition to s2 if event e (or e.foo, etc.) occurs and x is not equal to 1, and will go to s3 if any other event occurs.

```
<state id=s">
   <transition event="e" cond="x==1" target="s1"/>
   <transition event="e" target="s2"/>
   <transition event="*" target="s3"/>
</state>
```

3.1.2 Compound States

One of the most powerful concepts in Harel notation is the idea that states may have internal structure. In particular, a <state> element may contain nested <state> elements. Such a state is called a compound state and we speak of it as the parent state, while the nested elements are child states. The child states may themselves have nested children and the nesting may proceed to any depth. Ultimately we will reach a state that does not contain any child states. Such a state is called an atomic state. When a compound state is active, one and only one of its child states is active. Conversely, when an child state is active, its parent state must be active too. Thus at any point we have a set of active states, containing an atomic state and all of its ancestors. (We will see in the next section that multiple atomic states can be active at the same time.)

Compound states also affect how transitions are selected. When looking for transitions, the state machine first looks in the most deeply nested active state(s), i.e., in the atomic state(s) that have no substates. If no transitions match in the atomic state, the state machine will look in its parent state, then in the parent's parent, etc. Thus transitions in ancestor states serve as defaults that will be taken if no transition matches in a descendant state. If no transition matches in any state, the event is discarded.

3.1.3 Parallel States

The <parallel> element represents a state whose children execute in parallel. Like <state>, the <parallel> element contains <onentry>, <onexit>, <transition>, and <state> or <parallel> children. However, the semantics of <parallel> are different. When a <state> is active, exactly one of its children is active. When a <parallel> element is active, *all* of its children are active. Specifically, when the state machine enters the parent <parallel> state, it also enters each child state. The child states execute in parallel in the sense that any event that is processed is processed in each child state independently, and each child state may take a different transition in response to the event. (Similarly, one child state may take a transition in response to an event, while another child ignores it.) When all of the children reach final states, the <parallel> element itself is considered to be in a final state, and a completion event done.state.*id* is generated, where *id* is the id of the <parallel> element.

Transitions *within* the individual child elements operate normally. However whenever a transition is taken with a target *outside* the <parallel> element, the <parallel> element and all of its child elements are exited and the corresponding <onexit> handlers are executed. The handlers for the child elements execute first, in document order, followed by those of the parent <parallel> element, followed by an action expression in the <transition> element, and then the <onentry> handlers in the "target" state.

In the following example, parallel state 'p' has two children S1 and S2. Suppose a transition takes S1's child S12 as a target. (Note that this is permitted even though S12 is not the default initial state for S1 and that S11 is not, in fact, visited in the course of this example). Upon this transition, the state machine, in addition to entering S1 and S12, will also enter S1's parallel sibling S2 and its initial state S21. Once the transition has been taken, p, S1, S2, S12, and S21 will all be active. If event 'e1' occurs, it will cause S12 to transition to S1Final, and S21 to transition to S22. Entering S1Final will cause the event done.state.S1 to be generated. At this point, S1 is in a final state, but S2 is still active. Now suppose event 'e2' occurs. This will cause S22 to transition to S2Final, and the event done.state.S2 will be generated. Furthermore, since all of p's children are now in final states, the event 'done.state.p' will be generated, which will cause the transition contained in p to be triggered, exiting the entire region.

```
<parallel id="p">

    <transition event="done.state.p" target="someOtherState"/>
```

```
    <state id="S1" initial="S11">
        <state id="S11">
            <transition event="e4" target="S12"/>
        </state>
        <state id="S12">
            <transition event="e1" target="S1Final"/>
        </state>
        <final id="S1Final"/>
    </state>

    <state id="S2" initial="S21">
        <state id=S21">
            <transition event="e1" target="S22"/>
        </state>
        <state id="S22">
            <transition event="e2" target="S2Final/>
        </state>
        <final id="S2Final"/>
    </state>

</parallel>
```

Note that the semantics of the <parallel> element does not call for multiple threads or truly concurrent processing. The children of <parallel> execute in parallel in the sense that they are all simultaneously active and each one independently selects transitions for any event that is received. However, the parallel children process the event in a defined, serial order, so no conflicts or race conditions can occur. See *D Algorithm for SCXML Interpretation* for a detailed description of the semantics of <parallel> and the rest of SCXML.

3.1.4 Initial, Final, and History States

In the presence of compound states, transitions no longer simply move from the current active state to a new active state, but from one set of active states to another. (See *3.11 Legal State Configurations and Specifications* for details.) If the target of a transition is an atomic state, the state machine will enter not only the atomic state, but also any of its ancestor states that are not already active. Conversely, a transition may take a compound state as its target. In this case, one of the compound state's children must also become active, but the transition does not specify which one. In this case we look at the target state's *3.6 <initial>* child which specifies the state's default initial state, that is, the child state to enter if the transition does not specify one. (If the default initial state is itself compound, the state machine will also enter its default initial state, and so on recursively until it reaches an atomic state.) The presence of default initial states provides a form of encapsulation, since a transition may select a compound state as its target without understanding its internal substate structure.

The default initial state of a compound state may also be specified via the 'initial' attribute. The only difference between the <initial> element and the 'initial' attribute is that the <initial> element contains a <transition> element which may in turn contain executable content which will be executed before the default state is entered. If the 'initial' attribute is specified instead, the specified state will be entered, but no executable content will be executed. (If neither the <initial> child or the 'initial' element is specified, the default initial state is the first child state in document order.) As an example, suppose that parent state S contains child states S1 and S2 in that order. If S specifies S1 as its default initial state via the 'initial' attribute (or fails to specify any initial state), then any transition that specifies S as its target will result in the state machine entering S1 as well as S. In this case, the result is exactly the same as if the transition had taken S1 as its target. If, on the other hand, S specifies S1 as its default initial state via an <initial> element containing a <transition> with S1 as its target, the <transition> can contain executable content which will execute before the default entry into S1. In this case, there is a difference between a transition that takes S as its target and one that takes S1 as its target. In the former case, but not in the latter, the executable content inside the <initial> transition will be executed.

A compound state may also have final and history states as children. *3.7 <final>* is used to signify that the parent state is in some sense "done" with its processing. When a state machine enters a <final> substate of a compound state, the parent state remains active, but the event "done.state.*id*" is generated, where *id* is the state id of the parent state. This event can trigger a transition in any ancestor state (including the parent). If the transition takes a target outside the parent state, the "done.state.*id*" event in effect serves as a signal that it is time to leave the parent state. *3.10 <history>* allows for pause and resume semantics in compound states. Before the state machine exits a compound state, it records the state's active descendants. If the 'type' attribute of the <history> state is set to "deep", the state machine saves the state's full active descendant configuration, down to the atomic descendant(s). If 'type' is set to "shallow", the state machine remembers only which immediate child was active. After that, if a transition takes a <history> child of the state as its target, the state machine re-enters not only the parent compound state but also the state(s) in the saved configuration. Thus a transition with a deep history state as its target returns to exactly where the state was when it was last exited, while a transition with a shallow history state as a target re-enters the previously active child state, but will enter the child's default initial state (if the child is itself compound.)

3.1.5 'Type' and Transitions

In the case of a transition located in a compound state, the 'type' attribute is significant. The behavior of a transition with 'type' of "external" (the default) is defined in terms of the transition's source state (which is the state that contains the transition), the transition's target state(or states), and the Least Common Compound Ancestor (LCCA) of the source and target states (which is the closest compound state that is an ancestor of all the source and target states). When a transition is taken, the state machine will exit all active states that are proper descendants of the LCCA, starting with the innermost one(s) and working up to the immediate descendant(s) of the LCCA. (A 'proper descendant' of a state is a child, or a child of a child, or a child of a child of a child, etc.) Then the state machine enters the target state(s), plus any states that are between it and the LCCA, starting with the outermost one (i.e., the immediate descendant of the LCCA) and working down to the target state(s). As states are exited, their <onexit> handlers are executed. Then the executable content in the transition is executed, followed by the <onentry> handlers of the states that are entered. If the target state(s) of the transition is not atomic, the state machine will enter their default initial states recursively until it reaches an atomic state(s).

In the example below, assume that state s11 is active when event 'e' occurs. The source of the transition is state s1, its target is state s21, and the LCCA is state S. When the transition is taken, first state S11 is exited, then state s1, then state s2 is entered, then state s21. Note that the LCCA S is neither entered nor exited. For more details see *3.13 Selecting and Executing Transitions* and *D Algorithm for SCXML Interpretation*.

```
<state id="S" initial="s1">
   <state id="s1" initial="s11">
      <onexit>
         <log expr="'leaving s1'"/>
     </onexit>

     <state id="s11">
        <onexit>
           <log expr="'leaving s11'"/>
        </onexit>
```

```
      </state>

      <transition event="e" target="s21">
         <log expr="'executing transition'"/>
      </transition>

   </state>

   <state id="s2" initial="s21">
      <state id="s21">
         <onentry>
            <log expr="'entering s21'"/>
         </onentry>
      </state>
      <onentry>
         <log expr="'entering s2'"/>
      </onentry>
   </state>

   <onentry>
     <log expr="'entering S'"/>
   <onentry>
   <onexit>
     <log expr="'leaving S'"/>
   <onexit>
</state>

==== log output will be ======>

leaving s11
leaving s1
executing transition
entering s2
entering s21
```

The behavior of transitions with 'type' of "internal" is identical, except in the case of a transition whose source state is a compound state and whose target(s) is a descendant of the source. In such a case, an internal transition will not exit and re-enter its source state, while an external one will, as shown in the example below.

```
<state id="S" initial="s1">
   <state id="s1" initial="s11">
      <onentry>
        <log expr="entering S1"/>
        </onentry>
      <onexit>
         <log expr="'leaving s1'"/>
     </onexit>

     <state id="s11">
       <onentry>
        <log expr="entering s11"/>
       </onentry>
        <onexit>
           <log expr="'leaving s11'"/>
        </onexit>
     </state>

     <transition event="e" target="s11" type="internal">
        <log expr="'executing transition'"/>
     </transition>

   </state>



==== log output will be ======>

leaving s11
executing transition
entering s11

=== if transition were external, log output would be ====>

leaving s11
leaving s1
executing transition
entering s1
entering s11
```

If the 'target' on a <transition> is omitted, then the value of 'type' does not have any effect and taking the transition does not change the state configuration but does invoke the executable content that is included in the transition. Note that this is different from a <transition> whose 'target' is its source state. In the latter case, the state is exited and reentered, triggering execution of its <onentry> and <onexit> executable content.

3.2 <scxml>

[This section is normative.]

The top-level wrapper element, which carries version information. The actual state machine consists of its children. Note that only one of the children is active at any one time. See *3.11 Legal State Configurations and Specifications* for details.

3.2.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description

initial	false	none	IDREFS	none	A legal state specification. See *3.11 Legal State Configurations and Specifications* for details.	The id of the initial state(s) for the document. If not specified, the default initial state is the first child state in document order.
name	false	none	NMTOKEN	none	Any valid NMTOKEN	The name of this state machine. It is for purely informational purposes.
xmlns	true	none	URI	none	The value *must* be "http://www.w3.org/2005/07/scxml".	
version	true	none	decimal	none	The value *must* be "1.0"	
datamodel	false	none	NMTOKEN	platform-specific	"null", "ecmascript", "xpath" or other platform-defined values.	The datamodel that this document requires. "null" denotes the Null datamodel, "ecmascript" the ECMAScript datamodel, and "xpath" the XPath datamodel, as defined in *B Data Models*.
binding	false	none	enum	"early"	"early", "late"	The data binding to use. See *5.3.3 Data Binding* for details.

3.2.2 Children

- <state> A compound or atomic state. Occurs zero or more times. See *3.3 <state>* for details.
- <parallel> A parallel state. Occurs zero or more times. See *3.4 <parallel>* for details.
- <final> A top-level final state in the state machine. Occurs zero or more times. The SCXML processor *must* terminate processing when the state machine reaches this state. See *3.7 <final>* for details.
- <datamodel> Defines part or all of the data model. Occurs 0 or 1 times. See *5.2 <datamodel>*
- <script> Provides scripting capability. Occurs 0 or 1 times. *5.8 <script>*

A conformant SCXML document *must* have at least one <state>, <parallel> or <final> child. At system initialization time, the SCXML Processor *must* enter the states specified by the 'initial' attribute, if it is present. If it is not present, the Processor *must* enter the first state in document order. Platforms *should* document their default data model.

3.3 <state>

[This section is normative.]

Holds the representation of a state.

3.3.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
id	false	none	ID	none	A valid id as defined in [XML Schema]	The identifier for this state. See *3.14 IDs* for details.
initial	false	MUST NOT be specified in conjunction with the <initial> element. MUST NOT occur in atomic states.	IDREFS	none	A legal state specification. See *3.11 Legal State Configurations and Specifications* for details.	The id of the default initial state (or states) for this state.

3.3.2 Children

- <onentry> Optional element holding executable content to be run upon entering this <state>. Occurs 0 or more times. See *3.8 <onentry>*
- <onexit> Optional element holding executable content to be run when exiting this <state>. Occurs 0 or more times. See *3.9 <onexit>*
- <transition> Defines an outgoing transition from this state. Occurs 0 or more times. See *3.5 <transition>*
- <initial> In states that have substates, an optional child which identifies the default initial state. Any transition which takes the parent state as its target will result in the state machine also taking the transition contained inside the <initial> element. See *3.6 <initial>*
- <state> Defines a sequential substate of the parent state. Occurs 0 or more times.
- <parallel> Defines a parallel substate. Occurs 0 or more times. See *3.4 <parallel>*
- <final>. Defines a final substate. Occurs 0 or more times. See *3.7 <final>*.
- <history> A child pseudo-state which records the descendant state(s) that the parent state was in the last time the system transitioned *from* the parent. May occur 0 or more times. See *3.10 <history>*.
- <datamodel> Defines part or all of the data model. Occurs 0 or 1 times. See *5.2 <datamodel>*
- <invoke> Invokes an external service. Occurs 0 or more times. See *6.4 <invoke>* for details.

[Definition: An *atomic state* is a <state> that has no <state>, <parallel> or <final> children.]

[Definition: A *compound state* is a <state> that has <state>, <parallel>, or <final> children (or a combination of these).]

[Definition: The *default initial state(s)* of a compound state are those specified by the 'initial' attribute or <initial> element, if either is present. Otherwise it is the state's first child state in document order.]

In a conformant SCXML document, a compound state *may* specify either an "initial" attribute or an <initial> element, but not both. See *3.6 <initial>* for a discussion of the difference between the two notations.

3.4 <parallel>

[This section is normative.]

The <parallel> element encapsulates a set of child states which are simultaneously active when the parent element is active.

3.4.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
id	false		ID	none	A valid id as defined in [XML Schema]	The identifier for this state. See *3.14 IDs* for details.

3.4.2 Children

- <onentry> Holds executable content to be run upon entering the <parallel> element. Occurs 0 or more times. See *3.8 <onentry>*
- <onexit> Holds executable content to be run when exiting this element. Occurs 0 or more times. See *3.9 <onexit>*
- <transition> Defines an outgoing transition from this state. Occurs 0 or more times. See *3.5 <transition>*
- <state> Defines a parallel substate region. Occurs 0 or more times. See *3.3 <state>*.
- <parallel> Defines a nested set of parallel regions. Occurs 0 or more times.
- <history> A child which represents the state configuration that this state was in the last time the system transitioned *from* it. A transition with this history pseudo-state as its target is in fact a transition to the set of descendant states that were active the last time this state was exited. Occurs 0 or more times. See *3.10 <history>*.
- <datamodel> Defines part or all of the data model. Occurs 0 or 1 times. See *5.2 <datamodel>*
- <invoke> Invokes an external service. Occurs 0 or more times. See *6.4 <invoke>* for details.

3.5 <transition>

[This section is normative.]

Transitions between states are triggered by events and conditionalized via guard conditions. They may contain executable content, which is executed when the transition is taken.

3.5.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
event	false		EventsTypes.datatype.	none	A space-separated list of event descriptors. See *3.12.1 Event Descriptors* for details.	A list of designators of events that trigger this transition. See *3.13 Selecting and Executing Transitions* for details on how transitions are selected and executed. See *E Schema* for the definition of the datatype.
cond	false		Boolean expression	'true'	Any boolean expression. See *5.9.1 Conditional Expressions* for details.	The guard condition for this transition. See *3.13 Selecting and Executing Transitions* for details.
target	false	.	IDREFS	none	A legal state specification. See *3.11 Legal State Configurations and Specifications* for details.	The identifier(s) of the state or parallel region to transition to. See *3.13 Selecting and Executing Transitions* for details.
type	false		enum	"external"	"internal" "external"	Determines whether the source state is exited in transitions whose target state is a descendant of the source state. See *3.13 Selecting and Executing Transitions* for details.

3.5.2 Children

- The children of <transition> are executable content that is run after all the <onexit> handlers and before the all <onentry> handlers that are triggered by this transition. See *4 Executable Content*

A conformant SCXML document *must* specify at least one of 'event', 'cond' or 'target'. *3.13 Selecting and Executing Transitions* contains more detail on the semantics of transitions.

3.6 <initial>

[This section is normative.]

This element represents the default initial state for a complex <state> element (i.e. one containing child <state> or <parallel> elements.

3.6.1 Attribute Details

None

3.6.2 Children

- <transition> A transition whose 'target' specifies the default initial state(s). Occurs once. In a conformant SCXML document, this transition *must not* contain 'cond' or 'event' attributes, and *must* specify a non-null 'target' whose value is a valid state specification consisting solely of descendants of the containing state (see *3.11 Legal State Configurations and Specifications* for details). This transition *may* contain executable content.

3.7 <final>

[This section is normative.]

<final> represents a final state of an <scxml> or compound <state> element.

3.7.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
id	false		ID	none	A valid id as defined in [XML Schema]	The identifier for this state. See *3.14 IDs* for details.

3.7.2 Children

- <onentry> Optional element holding executable content to be run upon entering this state. Occurs 0 or more times. See *3.8 <onentry>* for details.
- <onexit> Optional element holding executable content to be run when exiting this state. Occurs 0 or more times. See *3.9 <onexit>* for details.
- <donedata> Optional element specifying data to be included in the done.state.*id* or done.invoke.*id* event. See *5.5 <donedata>* for details.

When the state machine enters the <final> child of a <state> element, the SCXML Processor *must* generate the event done.state.*id* after completion of the <onentry> elements, where *id* is the id of the parent state. Immediately thereafter, if the parent <state> is a child of a <parallel> element, and all of the <parallel>'s other children are also in final states, the Processor *must* generate the event done.state.*id* where *id* is the id of the <parallel> element.

When the state machine reaches the <final> child of an <scxml> element, it *must* terminate. See *D Algorithm for SCXML Interpretation* for details. If the SCXML session was triggered as the result by an <invoke> element in another session, the SCXML processor *must* generate the event done.invoke.*id* after termination and return it to the other session, where *id* is the unique identifier generated when the <invoke> element was executed. See *6.4 <invoke>* for details.

3.8 <onentry>

[This section is normative.]

A wrapper element containing executable content to be executed when the state is entered.

3.8.1 Attribute Details

None.

3.8.2 Children

The children of the <onentry> handler consist of executable content as defined in *4 Executable Content*.

The SCXML processor *must* execute the <onentry> handlers of a state in document order when the state is entered. In doing so, it *must* treat each handler as a separate block of executable content.

3.9 <onexit>

[This section is normative.]

A wrapper element containing executable content to be executed when the state is exited.

3.9.1 Attribute Details

None.

3.9.2 Children

The children of the <onexit> handler consist of executable content as defined in *4 Executable Content*.

The SCXML processor *must* execute the <onexit> handlers of a state in document order when the state is exited. In doing so, it *must* treat each handler as a separate block of executable content.

3.10 <history>

The <history> pseudo-state allows a state machine to remember its state configuration. A <transition> taking the <history> state as its target will return the state machine to this recorded configuration.

3.10.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
id	false		ID	none	A valid id as defined in [XML Schema]	Identifier for this pseudo-state. See *3.14 IDs* for details.
type	false		enum	"shallow"	"deep" or "shallow"	Determines whether the active atomic substate(s) of the current state or only its immediate active substate(s) are recorded.

3.10.2 Children

- <transition> A transition whose 'target' specifies the default history configuration. Occurs once. In a conformant SCXML document, this transition *must not* contain 'cond' or 'event' attributes, and *must* specify a non-null 'target' whose value is a valid state specification (see *3.11 Legal State Configurations and Specifications*). This transition *may* contain executable content. If 'type' is "shallow", then the 'target' of this <transition> *must* contain only immediate children of the parent state. Otherwise it *must* contain only descendants of the parent. Occurs once. (Note that under the definition of a legal state specification, if the parent of the history element is <state> and the default state specification contains a multiple states, then, in a conformant SCXML document, the 'type' of the history element *must* be "deep" and the states *must* be atomic descendants of a <parallel> element that is itself a descendant of the parent <state> element.)

If the 'type' of a <history> element is "shallow", the SCXML processor *must* record the immediately active children of its parent before taking any transition that exits the parent. If the 'type' of a <history> element is "deep", the SCXML processor *must* record the active atomic descendants of the parent before taking any transition that exits the parent. After the parent state has been visited for the first time, for each <history> element, we define the set of states that the processor has recorded to be the 'stored state configuration' for that history state. We also define the states specified by the 'target' of the <history> element's <transition> child to be the 'default stored state configuration' for that element. If a transition is executed that takes the <history> state as its target, the behavior depends on whether the parent state has been visited before. If it has, the SCXML processor *must* behave as if the transition had taken the stored state configuration for that history state as its target. If it has not, the SCXML processor *must* behave as if the transition had taken the default stored state configuration for that history state as its target. The Process *must* execute any executable content in the transition after the parent state's onentry handlers, and, in the case where the history pseudo-state is the target of an <initial> transition, the executable content inside the <initial> transition. (Note that in a conformant SCXML document, a <state> or <parallel> element *may* have both "deep" and "shallow" <history> children.)

3.11 Legal State Configurations and Specifications

[This section is normative.]

[Definition: A <state> or <parallel> element is *active* if it has been entered by a transition and has not subsequently been exited.]

[Definition: The *state configuration* of a state machine is the set of currently active states.]

An SCXML document places the state machine in an initial state configuration at initialization time (via the 'initial' attribute of the <scxml> element). Each transition that the state machine takes thereafter places the state machine in another state configuration (which need not be distinct from the former one.) A conformant SCXML document *must* place the state machine only in legal state configurations, where a legal state configuration is one that meets the following conditions:

- The configuration contains exactly one child of the <scxml> element.
- The configuration contains one or more atomic states.
- When the configuration contains an atomic state, it contains all of its <state> and <parallel> ancestors.
- When the configuration contains a non-atomic <state>, it contains one and only one of the state's children.
- If the configuration contains a <parallel> state, it contains all of its children.

It follows from this definition that if a state machine is in more than one atomic state, the atomic states can be traced back through a chain of <state> or <parallel> ancestors to a single <parallel> ancestor.

The 'target' attribute of a <transition> (or the 'initial' attribute of a <state> or <scxml> element) do not in the general case specify a full legal state configuration since 1) they can contain <parallel> or non-atomic <state> elements 2) they do not contain the ancestors of the states in the list. We therefore define a legal state specification to be a set of states such that 1) no state is an ancestor of any other state on the list, and 2) a full legal state configuration results when all ancestors and default initial descendants have been added. (Note that the process of adding default initial descendants is recursive, since the 'initial' value may itself be non-atomic.) In a conformant SCXML document, the value of an 'initial' attribute or the 'target' of a <transition> *must* either be empty or contain a legal state specification.

In a conformant SCXML document, there is an additional requirement on the value of the 'initial' attribute of a <state> and on the 'target' of a <transition> inside an <initial> or <history> element: all the states *must* be descendants of the containing <state> or <parallel> element.

3.12 SCXML Events

[This section is normative.]

Events are one of the basic concepts in SCXML since they drive most transitions. The internal structure of events is platform-specific as long as the following external interface is observed:

- The SCXML processor *must* make the data contained in an event accessible via the '_event' variable, as specified in *5.10 System Variables*.
- The SCXML processor *must* make the event's name accessible via the '_event' variable, as specified in *5.10 System Variables*. The SCXML processor *must* use this same name value to match against the 'event' attribute of transitions.

For the most part, the set of events raised during the execution of an SCXML document is application-specific and generated under author control by use of the <raise> and <send> elements. However, certain events are mandatory and generated automatically by the interpreter. These are described in *3.12.3 List of Errors and Events*. Platforms *may* extend the names of these automatically generated events by adding a suffix. For example, a platform could extend done.state.*id* with a timestamp suffix and generate done.state.*id.timestamp* instead. Because any prefix of done.state.*id* is also a prefix of done.state.*id.timestamp*, any transition that matches the former event will also match the latter.

3.12.1 Event Descriptors

Like an event name, an event descriptor is a series of alphanumeric characters segmented into tokens by the "." character. The 'event' attribute of a transition consists of one or more such event descriptors separated by spaces.

[Definition: A transition *matches* an event if at least one of its event descriptors matches the event's name.]

[Definition: An event descriptor *matches* an event name if its string of tokens is an exact match or a prefix of the set of tokens in the event's name. In all cases, the token matching is case sensitive.]

For example, a transition with an 'event' attribute of "error foo" will match event names "error", "error.send", "error.send.failed", etc. (or "foo", "foo.bar" etc.) but would not match events named "errors.my.custom", "errorhandler.mistake", "error.send" or "foobar".

For compatibility with CCXML, and to make the prefix matching possibly more clear to a reader of the SCXML document, an event descriptor *may* also end with the wildcard '.*', which matches zero or more tokens at the end of the processed event's name. Note that a transition with 'event' of "error", one with "error.", and one with "error.*" are functionally equivalent since they are token prefixes of exactly the same set of event names.

An event designator consisting solely of "*" can be used as a wildcard matching any sequence of tokens, and thus any event. Note that this is different from a transition lacking the 'event' attribute altogether. Such an eventless transition does not match any event, but will be taken whenever its 'cond' attribute evaluates to 'true'. As shown in *D Algorithm for SCXML Interpretation*, the SCXML interpreter will check for such eventless transitions when it first enters a state, before it looks for transitions driven by internal or external events.

3.12.2 Errors

Once the SCXML processor has begun executing a well-formed SCXML document, it *must* signal any errors that occur by raising SCXML events whose names begin with 'error.'. the processor *must* place these events in the internal event queue and *must* process them like any other event. (Note in particular, they are not processed immediately if there are other events in the queue and they are ignored if no transition is found that matches them.) Two error events are defined in this specification: 'error.communication' and 'error.execution'. The former cover errors occurring while trying to communicate with external entities, such as those arising from <send> and <invoke>, while the latter category consists of errors internal to the execution of the document, such as those arising from expression evaluation.

The set of error events may be extended in future versions of this specification. However, the set of names beginning with 'error.platform' is reserved for platform- and application-specific errors. Therefore applications and platforms *may* extend the set of errors defined in this specification in two ways. First by adding a suffix to an error name defined in this specification, and second by using 'error.platform' with or without a suffix. In addition, platforms *may* include additional information about the nature of the error in the 'data' field of the event. See *5.10 System Variables* for details.

Note however that authors can arrange for otherwise unhandled errors to cause the interpreter to exit by creating a transition with "event" attribute of 'error' and a target of any top-level final state (i.e. one that is a child of <scxml>). If such a transition T is placed in a state S, it will cause the state machine to terminate on any error that is raised in S or one of its substates and is not handled by another transition that is placed in a substate of S or in S and preceding T in document order.

3.12.3 List of Errors and Events

The following events are generated automatically by the SCXML implementation under conditions defined elsewhere in this document.

Name	Description	Defined in	See also
done.state.*id*	Indicates that the state machine has entered a final substate of state *id*.	*3.7 <final>*	*3.1 Introduction*
done.invoke.*id*	Indicates that the invoked process with invokeid *id* has completed processing.	*6.4 <invoke>*	*3.7 <final>*, exitInterpreter procedure in *D Algorithm for SCXML Interpretation*
error.communication	Indicates that an error has occurred while trying to communicate with an external entity.	*3.12.2 Errors*	*6.2 <send>*, *C.1 SCXML Event I/O Processor*, *C.2 Basic HTTP Event I/O Processor*
error.execution	Indicates that an error internal to the execution of the document has occurred, such as one arising from expression evaluation.	*3.12.2 Errors*	*4.6 <foreach>*, *5.4 <assign>*, *5.7 <param>*, *5.9.1 Conditional Expressions*, *5.9.2 Location Expressions*, *5.9.3 Legal Data Values and Value Expressions*, *5.9.4 Errors in Expressions*, *5.10 System Variables*, *6.2 <send>*, *B.2.4 Location Expressions*, *B.2.7 <assign>*
error.platform	Indicates that a platform- or application-specific error has occurred.	*3.12.2 Errors*	

3.13 Selecting and Executing Transitions

[This section is normative.]

To simplify the following definitions, we introduce the event NULL. NULL has no name and is used only in these definitions. It never occurs in the event queues of an SCXML Processor. All other events have names and are distinct from NULL. (In effect, NULL is a pseudo-event that is used in these definitions as a trigger for eventless transitions.)

[Definition: A transition T is *enabled* by named event E in atomic state S if a) T's source state is S or an ancestor of S,and b) T matches E's name (see *3.12.1 Event Descriptors*) and c) T lacks a 'cond' attribute or its 'cond' attribute evaluates to "true". A transition is *enabled* by NULL in atomic state S if a) T lacks an 'event' attribute, and b) T's source state is S or an ancestor of S and c) T lacks an 'cond' attribute or its 'cond' attribute evaluates to "true". (Note that such a transition can never be enabled by any named event.)]

[Definition: The *source state* of a transition is the <state> or <parallel> element that it occurs in. The *effective target state(s)* of the transition is the state or set of states specified by its 'target' attribute, with any history states being replaced by the corresponding stored state configuration or default stored state configuration. The *complete target set* of a transition consists of all the states that will be active after the transition is taken. It contains the effective target states of the transition plus all their ancestors, expanded by the recursive application of the following two operations: 1) if any <parallel> element is a member of the set, any of its children that are not members of the set must be added 2) if any compound <state> is in the set and none of its children is in the set, its default initial state(s) are added to the set. Any state whose child(ren) are added to the complete target set by clause 2 is called a *default entry state.*]

[Definition: The *exit set* of a transition in configuration C is the set of states that are exited when the transition is taken when the state machine is in C. If the transition does not contain a 'target', its exit set is empty. Otherwise (i.e., if the transition contains a 'target'), if its 'type' is "external", its exit set consists of all active states in C that are proper descendants of the Least Common Compound Ancestor (LCCA) of the source and target states. Otherwise, if the transition has 'type' "internal", its source state is a compound state, and all its target states are proper descendants of its source state, the exit set consists of all active states in C that are proper descendants of its source state. (If a transition has 'type' of "internal", but its source state is not compound or its target states are not all proper descendants of its source state, its exit set is defined as if it had 'type' of "external". The exit set of a set of transitions is the union of the exit sets of the individual transitions.]

[Definition: The *entry set* of a transition in configuration C is the set of states that are entered when the transition is taken. If a transition does not contain a 'target', its entry set is empty. Otherwise, it consists of all members of the transition's complete target set that that are not currently active or are in the exit set. (Thus the entry set consists of all members of the transition's complete target set that will not be active once the states in the exit set have been exited.) The entry set of a set of transitions is the union of the entry sets of the individual transitions.]

[Definition: A transition T is *optimally enabled* by event E in atomic state S if a) T is enabled by E in S and b) no transition that precedes T in document order in T's source state is enabled by E in S and c) no transition is enabled by E in S in any descendant of T's source state.]

[Definition: Two transitions T1 and T2 *conflict* in state configuration C if their exit sets in C have a non-null intersection.]

N.B. If two transitions conflict, then taking them both may lead to an illegal configuration. Hence, only one of the transitions may safely be taken. In order to resolve conflicts between transitions, we assign priorities to transitions as follows: let transitions T1 and T2 conflict, where T1 is optimally enabled in atomic state S1, and T2 is optimally enabled in atomic state S2, where S1 and S2 are both active. We say that T1 has a higher priority than T2 if a) T1's source state is a descendant of T2's source state, or b) S1 precedes S2 in document order.

[Definition: The *optimal transition set* enabled by event E in state configuration C is the largest set of transitions such that a) each transition in the set is optimally enabled by E in an atomic state in C b) no transition conflicts with another transition in the set c) there is no optimally enabled transition outside the set that has a higher priority than some member of the set.]

[Definition: A *microstep* consists of the execution of the transitions in an optimal enabled transition set.]

[Definition: A *macrostep* is a series of one or more microsteps ending in a configuration where the internal event queue is empty and no transitions are enabled by NULL.]

To execute a microstep, the SCXML Processor *must* execute the transitions in the corresponding optimal enabled transition set. To execute a set of transitions, the SCXML Processor *must* first exit all the states in the transitions' exit set in exit order. It *must* then execute the executable content contained in the transitions in document order. It *must* then enter the states in the transitions' entry set in entry order.

To exit a state, the SCXML Processor *must* execute the executable content in the state's <onexit> handler. Then it *must* cancel any ongoing invocations that were triggered by that state. Finally, the Processor *must* remove the state from the active state's list.

To enter a state, the SCXML Processor *must* add the state to the active state's list. Then it *must* execute the executable content in the state's <onentry> handler. If the state is a default entry state and has an <initial> child, the SCXML Processor *must* then execute the executable content in the <initial> child's <transition>.

At startup, the SCXML Processor *must* place the state machine in the configuration specified by the 'initial' attribute of the <scxml> element.

After entering the initial configuration, and after executing each microstep, the SCXML Processor *must* check the state configuration for <final> states that it has entered during the microstep. If it has entered a <final> state that is a child of <scxml>, it *must* halt processing. If it has entered a <final> state that is a child of a compound state, it *must* generate the event done.state.*id*, where *id* is the id of the compound state. If the compound state is itself the child of a <parallel> element, and all the <parallel> element's other children are in final states, the Processor *must* generate the event done.state.*id*, where *id* is the id of the <parallel> elements.

After checking the state configuration, the Processor *must* select the optimal transition set enabled by NULL in the current configuration. If the set is not empty, it *must* execute it as a microstep. If the set is empty, the Processor *must* remove events from the internal event queue until the queue is empty or it finds an event that enables a non-empty optimal transition set in the current configuration. If it finds such a set, the processor *must* then execute it as a microstep. (Otherwise the internal event queue is empty and the Processor has completed a macrostep.)

After completing a macrostep, the SCXML Processor *must* execute in document order the <invoke> handlers in all states that have been entered since the completion of the last macrostep. Then the Processor *must* remove events from the external event queue, waiting till events appear if necessary, until it finds one that enables a non-empty optimal transition set in the current configuration. The Processor *must* then execute that set as a microstep.

3.14 IDs

[This section is normative.]

In a conformant SCXML document, the values of all attributes of type "id" *must* be unique within the session. When such an attribute is defined to be optional and the author omits it, then, for elements other than <send> and <invoke>, the SCXML processor *must* generate a unique id automatically at document load time. (Note that Such system generated IDs cannot normally be referenced elsewhere in the document because they are not known to the author. In particular, a state with a system generated ID cannot be the target of a transition.) The ids for <send> and <invoke> are subtly different. In a conformant SCXML document, they *must* be unique within the session, but in the case where the author does not provide them, the processor *must* generate a new unique ID not at load time but *each time the element is executed*. Furthermore the attribute 'idlocation' can be used to capture this automatically generated id. Finally note that the automatically generated id for <invoke> has a special format. See *6.4.1 Attribute Details* for details. The SCXML processor *may* generate all other ids in any format, as long as they are unique.

4 Executable Content

4.1 Introduction

[This section is informative.]

Executable content allows the state machine to *do* things. It provides the hooks that allow an SCXML session to modify its data model and interact with external entities. Executable content consists of actions that are performed as part of taking transitions. In particular, executable content occurs inside <onentry> and <onexit> elements as well as inside transitions. When the state machine takes a transition, it executes

the <onexit> executable content in the states it is leaving, followed by the content in the transition, followed by the <onentry> content in the states it is entering.

This standard defines elements of executable content which can raise events*4.2 <raise>*, communicate with external entities *6.2 <send>*, log information *4.7 <log>* execute scripts *5.8 <script>* and modify the data model *5.4 <assign>*, as well as control constructs to conditionalize execution *4.3 <if>* and to iterate over the items in a collection *4.6 <foreach>*. In addition, SCXML implementations are allowed to define their own, platform-specific executable content (see *4.10 Extensibility of Executable Content*).

4.2 <raise>

[This section is normative.]

The <raise> element raises an event in the current SCXML session. Note that the event will not be processed until the current block of executable content has completed and all events that are already in the internal event queue have been processed. For example, suppose the <raise> element occurs first in the <onentry> handler of state S followed by executable content elements ec1 and ec2. If event e1 is already in the internal event queue when S is entered, the event generated by <raise> will not be processed until ec1 and ec2 have finished execution and e1 has been processed.

4.2.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
event	true		NMTOKEN	none		Specifies the name of the event. This will be matched against the 'event' attribute of transitions.

4.2.2 Children

None.

The SCXML processor *must* place the event that is generated at the rear of the session's internal event queue.

4.3 <if>

[This section is normative.]

<if> is a container for conditionally executed elements.

4.3.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
cond	true		Conditional expression	none	A valid conditional expression	A boolean expression. See *5.9.1 Conditional Expressions* for details.

4.3.2 Children

- <elseif> Occurs 0 or more times. See *4.4 <elseif>*
- <else> Occurs 0 or 1 times. See *4.5 <else>*
- The other children of <if> consist of executable content. Note that since <if> itself is executable content, nested <if> statements are allowed.

The behavior of <if> is defined in terms of partitions of executable content. The first partition consists of the executable content between the <if> and the first <elseif>, <else> or </if> tag. Each <elseif> tag defines a partition that extends from it to the next <elseif>, <else> or </if> tag. The <else> tag defines a partition that extends from it to the closing </if> tag. In a conformant SCXML document, a partition *may* be empty. In a conformant SCXML document, <else> *must* occur after all <elseif> tags.

When the <if> element is executed, the SCXML processor *must* execute the first partition in document order that is defined by a tag whose 'cond' attribute evaluates to true, if there is one. Otherwise, it *must* execute the partition defined by the <else> tag, if there is one. Otherwise it *must not* execute any of the executable content.

Here is an example:

```
<if cond="cond1">
  <!-- selected when "cond1" is true -->
<elseif cond="cond2"/>
  <!-- selected when "cond1" is false and "cond2" is true -->
<elseif cond="cond3"/>
  <!-- selected when "cond1" and "cond2" are false and "cond3" is true -->
<else/>
  <!-- selected when "cond1", "cond2", and "cond3" are false -->
</if>
```

4.4 <elseif>

[This section is normative.]

4.4.1 Overview

<elseif> is an empty element that partitions the content of an <if>, and provides a condition that determines whether the partition is executed.

4.5 <else>

[This section is normative.]

4.5.1 Overview

<else> is an empty element that partitions the content of an <if>. It is equivalent to an <elseif> with a "cond" that always evaluates to true.

4.5.2 Attribute Details

None.

4.6 <foreach>

[This section is normative.]

4.6.1 Overview

The <foreach> element allows an SCXML application to iterate through a collection in the data model and to execute the actions contained within it for each item in the collection.

4.6.2 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
array	true		Value expression	none	A value expression that evaluates to an iterable collection.	The <foreach> element will iterate over a shallow copy of this collection.
item	true		xsd:string	none	Any variable name that is valid in the specified data model.	A variable that stores a different item of the collection in each iteration of the loop.
index	false		xsd:string	none	Any variable name that is valid in the specified data model.	A variable that stores the current iteration index upon each iteration of the foreach loop.

4.6.3 Children

The children of <foreach> consist of one or more items of executable content. (Note that they are considered to be part of the same block of executable content as the parent <foreach> element.)

The SCXML processor *must* declare a new variable if the one specified by 'item' is not already defined. If 'index' is present, the SCXML processor *must* declare a new variable if the one specified by 'index' is not already defined. If 'array' does not evaluate to a legal iterable collection, or if 'item' does not specify a legal variable name, the SCXML processor *must* terminate execution of the <foreach> element and the block that contains it, and place the error error.execution on the internal event queue.

The SCXML processor *must* act as if it has made a shallow copy of the collection produced by the evaluation of 'array'. Specifically, modifications to the collection during the execution of <foreach> *must not* affect the iteration behavior. The SCXML processor *must* start with the first item in the collection and proceed to the last item in the iteration order that is defined for the collection. (This order depends on the data model in use.) For each item in turn, the processor *must* assign it to the item variable. (Note that the assigned value *may* be null or undefined if the collection contains a null or undefined item.) After making the assignment, the SCXML processor *must* evaluate its child executable content. It *must* then proceed to the next item in iteration order. If the evaluation of any child element causes an error, the processor *must* cease execution of the <foreach> element and the block that contains it. (Note that SCXML does not provide break functionality to interrupt <foreach>, however targetless and/or eventless transitions can provide sophisticated iterative behavior within the SCXML application itself.)

4.7 <log>

[This section is normative.]

4.7.1 Overview

<log> allows an application to generate a logging or debug message.

4.7.2 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
label	false		string	empty string		A character string with an implementation-dependent interpretation. It is intended to provide meta-data about the log string specified by 'expr'.
expr	false		Value expression	none		An expression returning the value to be logged. See *5.9.3 Legal Data Values and Value Expressions* for details. The nature of the logging mechanism is implementation-dependent. For example, the SCXML processor may convert this value to a convenient format before logging it.

4.7.3 Children

None.

The manner in which the message is displayed or logged is platform-dependent. The SCXML processor *must* ensure that <log> has no side-effects on document interpretation.

4.8 Other Executable Content

[This section is normative.]

The following elements of executable content are defined elsewhere in this specification. They *may* occur wherever executable content is allowed and *must not* occur anyplace else.

- <assign>. Changes the value of a location in the data model. See *5.4 <assign>* for details.
- <script>. Provides scripting capabilities. See *5.8 <script>* for details.
- <send>. Sends an event to a specified destination. See *6.2 <send>* for details.
- <cancel>. Cancels an event that was to be sent See *6.3 <cancel>* for details.

4.9 Evaluation of Executable Content

[This section is normative.]

Wherever executable content is permitted, an arbitrary number of elements *may* occur. Such a sequence of elements of executable content is called a block. For example, if transition t takes the state machine from atomic state S1 to atomic state S2, there are three blocks of executable content executed: the one in the <onexit> handler of S1, the one inside t, and the one inside the <onentry> handler of S2. The SCXML processor *must* execute the elements of a block in document order. If the processing of an element causes an error to be raised, the processor *must not* process the remaining elements of the block. (The execution of other blocks of executable content is not affected.)

Events raised during the processing of executable content are treated like any other events. Note in particular, that error events will not be removed from the queue and processed until all events preceding them in the queue have been processed. See *3.12.2 Errors*. for details.

4.10 Extensibility of Executable Content

[This section is normative.]

Implementations *may* provide additional executable content corresponding to special features of their implementations. The functionality of such platform-specific content is not restricted, except that it *must not* cause transitions or any form of change of state (except indirectly, by raising events that trigger transitions). Note that SCXML treats the executable content triggered by a transition as a single blocking operation and that no events are processed until all the executable content has completed. For example, when taking a transition into state S, the SCXML processor will not process any events or take any transitions until all <onentry> handlers in S have finished. It is thus important that all executable content, including platform-specific extensions, execute swiftly.

In a conformant SCXML document any extensions to executable content *must not* be defined the 'scxml' namespace. (Note that the schema *E Schema* allows elements from arbitrary namespaces inside blocks of executable content.) The following example shows the incorporation of CCXML functionality (see [CCXML 1.0]) into SCXML. In particular an <accept> element in the 'ccxml' namespace is invoked as executable content inside a transition.

```
<transition event="ccxml:connection.alerting">
  <ccxml:accept connectionid="_event.data.connectionid"/>
</transition>
```

This markup is legal on any SCXML interpreter, but the behavior of <accept> element is platform-dependent. See *A.2 Conforming Processors* for details.

A general method for implementing extensions using the <send> element is presented in *G.7 Custom Action Elements*.

5 Data Model and Data Manipulation

5.1 Introduction

[This section is informative.]

The Data Model offers the capability of storing, reading, and modifying a set of data that is internal to the state machine. This specification does not mandate any specific data model, but instead defines a set of abstract capabilities that can be realized by various languages, such as ECMAScript or XML/XPath. Implementations may choose the set of data models that they support. In addition to the underlying data structure, the data model defines a set of expressions as described in *5.9 Expressions*. These expressions are used to refer to specific locations in the data model, to compute values to assign to those locations, and to evaluate boolean conditions. Finally, the data model includes a set of system variables, as defined in *5.10 System Variables*, which are automatically maintained by the SCXML processor.

The data model is defined via the *5.2 <datamodel>* element, which contains zero or more *5.3 <data>* elements, each of which defines a single data element and assigns an initial value to it. These values may be specified in-line or loaded from an external source. They can then be updated via the *5.4 <assign>* element. The *5.5 <donedata>*, *5.6 <content>* , and *5.7 <param>* elements can be used to incorporate data into communications with external entities. Finally, the *5.8 <script>* element permits the incorporation of a scripting language.

The interpretation of these elements depends on the data model in question, and not all elements are supported in all data models. For the details of specific data models, see *B Data Models*.

5.2 <datamodel>

[This section is normative.]

<datamodel> is a wrapper element which encapsulates any number of <data> elements, each of which defines a single data object. The exact nature of the data object depends on the data model language used.

5.2.1 Attribute Details

None.

5.2.2 Children

- <data> Occurs 0 or more times. Each instance defines a named data element.

5.3 <data>

[This section is normative.]

The <data> element is used to declare and populate portions of the data model.

5.3.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
id	true		ID	none		The name of the data item. See *3.14 IDs* for details.
src	false		URI	none		Gives the location from which the data object should be fetched. See *5.9.3 Legal Data Values and Value Expressions* for details.
expr	false		Expression	none	Any valid value expression	Evaluates to provide the value of the data item. See *5.9.3 Legal Data Values and Value Expressions* for details.

5.3.2 Children

The children of the <data> element represent an in-line specification of the value of the data object.

In a conformant SCXML document, a <data> element *may* have either a 'src' or an 'expr' attribute, but *must not* have both. Furthermore, if either attribute is present, the element *must not* have any children. Thus 'src', 'expr' and children are mutually exclusive in the <data> element.

The SCXML Processor *must* use any values provided by the environment at instantiation time in place of those contained in the top-level <data> elements. (Top-level data elements are those that are children of the <datamodel > element that is a child of <scxml>). The manner in which the environment specifies these overriding values is platform-dependent.

If the 'expr' attribute is present, the Platform *must* evaluate the corresponding expression at the time specified by the 'binding' attribute of <scxml> and *must* assign the resulting value as the value of the data element. If the 'src' attribute is present, the Platform *must* fetch the specified object at the time specified by the 'binding' attribute of <scxml> and *must* assign it as the value of the data element. If child content is specified, the Platform *must* assign it as the value of the data element at the time specified by the 'binding' attribute of <scxml>. Note that in the latter two cases, the interpretation of the object or content will depend on the data model. See *B Data Models* for details. If the value specified for a <data> element (by 'src', children, or the environment) is not a legal data value, the SCXML Processor *must* raise place error.execution in the internal event queue and *must* create an empty data element in the data model with the specified id.

Implementations *may* predeclare and predefine variables in the data model. However, conformant SCXML implementations *should not* assume the existence of any predeclared or predefined variables (i.e., ones not explicitly defined by <data>).

Note that this specification does not define any way to modify the data model except by<assign>, <finalize>, and possibly platform-specific elements of executable content. In particular, no means is defined for external entities to modify the data model. In this sense the data model is local to the SCXML session and the SCXML Processor checks for eventless transitions (i.e. ones that are triggered based only on the state of the data model) only after entering a state or processing an event. However in some deployments it may be possible for external entities to modify the data model. For example, if SCXML is implemented in JavaScript in a browser, the scope of a document's data model is always accessible through the main window object and thus JavaScript code elsewhere in the window can modify the data model independent of the SCXML interpretation algorithm. Such a situation can lead to race conditions and unpredictable behavior

5.3.3 Data Binding

Authors control when the initial values are assigned to the data elements by means of the 'binding' attribute on the <scxml> element. When 'binding' is assigned the value "early" (the default), the SCXML Processor *must* create all data elements and assign their initial values at document initialization time. When 'binding' is assigned the value "late", the SCXML Processor *must* create the data elements at document initialization time, but *must* assign the specified initial value to a given data element only when the state that contains it is entered for the first time, before any <onentry> markup. (The value of the data element between the time it is created and the time its parent state is first entered will depend on the data language chosen. The initial value specified by 'expr', 'src' or in-line content will be assigned to the data element even if the element already has a non-null value when the parent state is first entered.)

5.4 <assign>

[This section is normative.]

The <assign> element is used to modify the data model.

5.4.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
location	true		path expression	none	Any valid location expression.	The location in the data model into which to insert the new value. See *5.9.2 Location Expressions* for details.
expr	false	This attribute must not occur in an <assign> element that has children.	value expression	none	Any valid value expression	An expression returning the value to be assigned. See *5.9.3 Legal Data Values and Value Expressions* for details.

5.4.2 Children

The children of the <assign>element provide an in-line specification of the legal data value (see *5.9.3 Legal Data Values and Value Expressions*) to be inserted into the data model at the specified location.

A conformant SCXML document *must* specify either "expr" or children of <assign>, but not both.

Assignment to a data model is done by using a location expression to denote the part of the data model where the insertion is to be made. If the location expression does not denote a valid location in the data model or if the value specified (by 'expr' or children) is not a legal value for the location specified, the SCXML Processor *must* place the error 'error.execution' in the internal event queue. Otherwise, the SCXML Processor *must* place the specified value at the specified location. Note that the nature of the insertion and the definition of a legal value depends on the data model language used. Note also that data models *may* support additional attributes for <assign> beyond those specified here. See *B Data Models* for details.

5.5 <donedata>

[This section is normative.]

A wrapper element holding data to be returned when a <final> state is entered.

5.5.1 Attribute Details

None.

5.5.2 Children

- <content>. Specifies data to include in the event. May occur 0 or 1 times. See *5.6 <content>* .
- <param> Extracts data from the data model to include in the event. See *5.7 <param>* for details. May occur 0 or more times.

A conformant SCXML document *must* specify either a single <content> element or one or more <param> elements as children of <donedata>, but not both.

In cases where the SCXML Processor generates a 'done' event upon entry into the final state, it *must* evaluate the <donedata> elements <param> or <content> children and place the resulting data in the _event.data field. The exact format of that data will be determined by the data model (see *B Data Models* for details). In other cases (namely when the <final> element is a child of <scxml> and the state machine has not been triggered by <invoke>), the SCXML Processor *should* return the data to the environment in an implementation-dependent manner.

5.6 <content>

[This section is normative.]

A container element holding data to be passed to an external service.

5.6.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
expr	false	must not occur with child content	Value expression	none	Any valid value expression	A value expression. See *5.9.3 Legal Data Values and Value Expressions* for details.

5.6.2 Children

A conformant SCXML document *must not* specify both the 'expr' attribute and child content. When present, the children of <content> *may* consist of text, XML from any namespace, or a mixture of both.

The use of the <content> element depends on the context in which it occurs. See *5.5 <donedata>*, *6.2 <send>* and *6.4 <invoke>* for details. When the SCXML Processor evaluates the <content> element, if the 'expr' value expression is present, the Processor *must* evaluate it and use the result as the output of the <content> element. If the evaluation of 'expr' produces an error, the Processor *must* place error.execution in the internal event queue and use the empty string as the value of the <content> element. If the 'expr' attribute is not present, the Processor *must* use the children of <content> as the output. The interpretation of the output of the <content> element depends on the data model. See *B Data Models* for details. For the use of namespaces inside <content>, see *G.6 Inline Content and Namespaces*.

5.7 <param>

[This section is normative.]

The <param> tag provides a general way of identifying a key and a dynamically calculated value which can be passed to an external service or included in an event.

5.7.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
name	true		NMTOKEN	none	A string literal	The name of the key.
expr	false	May not occur with 'location'	value expression	none	Valid value expression	A value expression (see *5.9.3 Legal Data Values and Value Expressions*) that is evaluated to provide the value.
location	false	May not occur with 'expr'	location expression	none	Valid location expression	A location expression (see *5.9.2 Location Expressions*) that specifies the location in the datamodel to retrieve the value from.

A conformant SCXML document *must* specify either the 'expr' attribute of <param> or the 'location' attribute, but *must not* specify both. If the 'location' attribute does not refer to a valid location in the data model, or if the evaluation of the 'expr' produces an error, the SCXML Processor *must* place the error 'error.execution' on the internal event queue and *must* ignore the name and value. Otherwise the use of the name and value depends on the context in which the <param> element occurs. See *5.5 <donedata>*, *6.2 <send>* and *6.4 <invoke>* for details.

5.7.2 Children

None.

5.8 <script>

[This section is normative.]

The <script> element adds scripting capability to the state machine.

5.8.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
src	false	May not occur if the element has children.		none	A valid URI	Gives the location from which the script should be downloaded.

5.8.2 Children

The child content of the <script> element represents the script code to be executed.

A conformant SCXML document *must* specify either the 'src' attribute or child content, but not both. If 'src' is specified, the SCXML Processor *must* download the script from the specified location at load time. If the script can not be downloaded within a platform-specific timeout interval, the document is considered non-conformant, and the platform *must* reject it.

The SCXML Processor *must* evaluate any <script> element that is a child of <scxml> at document load time. It *must* evaluate all other <script> elements as part of normal executable content evaluation.

In a conformant SCXML document, the name of any script variable *may* be used as a location expression (see *5.9.2 Location Expressions*).

For an example of a data model incorporating scripting, see *B.2 The ECMAScript Data Model*.

5.9 Expressions

[This section is normative.]

SCXML contains three types of expressions, as described below. Different data models will support different languages for these expression types, but certain properties of the expressions are constant across languages and are defined here.

When "late" data binding is used, accessing data substructure in expressions before the corresponding <data> element is loaded *must* yield the same execution-time behavior as accessing non-existent data substructure in a loaded <data> instance. Such behavior is defined by the data expression language in use.

5.9.1 Conditional Expressions

Conditional expressions are used inside the 'cond' attribute of <transition>, <if> and <elseif>. If a conditional expression cannot be evaluated as a boolean value ('true' or 'false') or if its evaluation causes an error, the SCXML Processor *must* treat the expression as if it evaluated to 'false' and *must* place the error 'error.execution' in the internal event queue. The set of operators in conditional expressions varies depending on the data model, but all data models *must* support the 'In()' predicate, which takes a state ID as its argument and returns true if the state machine is in that state. This predicate allows coordination among parallel regions. Conditional expressions in conformant SCXML documents *should not* have side effects.

5.9.2 Location Expressions

Location expressions are used to specify a location in the data model, e.g. as part of the <assign>, <param>, <send> or <invoke> elements. The exact nature of a location depends on the data model. If a location expression cannot be evaluated to yield a valid location, the SCXML processor *must* place the error 'error.execution' in the internal event queue.

5.9.3 Legal Data Values and Value Expressions

A data model definition contains a specification of the underlying data structure. Such a specification of the data structure implicitly defines a set of "legal data values", namely the objects that can be part of such a data structure. In conjunction with this, the data model definition specifies a set of value expressions which can be evaluated at runtime to return legal data values. If a value expression does not return a legal data value, the SCXML Processor *must* place the error 'error.execution' in the internal event queue.

5.9.4 Errors in Expressions

The SCXML Processor *may* reject documents containing syntactically ill-formed expressions at document load time, or it *may* wait and place 'error.execution' in the internal event queue at runtime when the expressions are evaluated. If the processor waits until it evaluates the expressions at runtime to raise errors, it *must* raise errors caused by expressions returning illegal values at the points at which the expressions are to be evaluated. Note that this requirement holds even if the implementation is optimizing expression evaluation.

5.10 System Variables

[This section is normative.]

The SCXML Processor *must* maintain a protected portion of the data model containing information that can be useful to applications. We refer to the items in this special part of the data model as 'system variables'. Implementations *must* provide the following system variables, and *may* support others.

- *_event*. The SCXML Processor *must* use the variable '_event' to hold a structure containing the current event's name and any data contained in the event (see *5.10.1 The Internal Structure of Events*. The exact nature of the structure depends on the data model being used. See *B Data Models* for details. The SCXML Processor *must* bind the _event variable when an event is pulled off the internal or external event queue to be processed, and *must* keep the variable bound to that event until another event is processed. (It follows that when an application is testing the 'cond' attribute of a <transition> element that contains an 'event' attribute, _event will be bound to the event that the transition is being matched against. If the transition is selected to be executed, _event will remain bound to that event in the <onexit> handlers of the states being exited, the executable content of the transition itself, and the <onentry> handlers of the states being entered. In the case of <transition> elements that do not contain an 'event' attribute and the <onexit> and <onentry> handlers of any states that are exited or entered by such transitions, the _event variable will not have a easily predictable value since the transition is not being driven by an event. In these cases, _event will be bound to the last event that was matched against a transition.) The SCXML Processor *must not* bind _event at initialization time until the first event is processed. Hence _event is unbound when the state machine starts up. If the data in the event is not a legal instance of the data model language, and the Processor cannot translate it into one, then the Processor *must* place the error 'error.execution' in the internal event queue at the point at which it attempts to bind _event. In this case, the Processor *must* leave the event data part of the _event structure unbound. (Note that the event's name will still be available, however and that processing of both the original event and the error event will proceed as usual.)
- *_sessionid*. The SCXML Processor *must* bind the variable _sessionid at load time to the system-generated id for the current SCXML session. (This is of type NMTOKEN.) The Processor *must* keep the variable bound to this value until the session terminates.
- *_name*. The SCXML Processor *must* bind the variable _name at load time to the value of the 'name' attribute of the <scxml> element. The Processor *must* keep the variable bound to this value until the session terminates.
- *_ioprocessors*. The SCXML Processor *must* bind the variable _ioprocessors to a set of values, one for each Event I/O Processor that it supports. The syntax to access it depends on the data model. See *B Data Models* for details. The nature of the values associated with the individual Event I/O Processors depends on the Event I/O Processor in question. See *C Event I/O Processors* for details. The Processor *must* keep the variable bound to this set of values until the session terminates.
- *_x*. The variable _x is the root element for platform-specific system variables. The Processor *must* place all platform-specific system variables underneath it. The exact structure of the platform-specific variables depends on the data model. For example, in the ECMAScript data model *B.2 The ECMAScript Data Model*, '_x' will be a top-level ECMAScript object and the platform-specific system variables will be its properties.

The set of system variables may be expanded in future versions of this specification. Variable names beginning with '_' are reserved for system use. A conformant SCXML document *must not* contain ids beginning with '_' in the <data> element. Platforms *must* place all platform-specific system variables under the '_x' root.

The concrete realization of these variables in a specific data model depends on the language used. The Processor *must* cause any attempt to change the value of a system variable to fail and *must* place the error 'error.execution' on the internal event queue when such an attempt is made.

5.10.1 The Internal Structure of Events

Events have an internal structure which is reflected in the _event variable. This variable can be accessed to condition transitions (via boolean expressions in the 'cond' attribute) or to update the data model (via <assign>), etc.

The SCXML Processor *must* ensure that the following fields are present in all events, whether internal or external.

- *name*. This is a character string giving the name of the event. The SCXML Processor *must* set the name field to the name of this event. It is what is matched against the 'event' attribute of <transition>. Note that transitions can do additional tests by using the value of this field inside boolean expressions in the 'cond' attribute.
- *type*. This field describes the event type. The SCXML Processor *must* set it to: "platform" (for events raised by the platform itself, such as error events), "internal" (for events raised by <raise> and <send> with target '_internal') or "external" (for all other events).
- *sendid*. If the sending entity has specified a value for this, the Processor *must* set this field to that value (see *C Event I/O Processors* for details). Otherwise, in the case of error events triggered by a failed attempt to send an event, the Processor *must* set this field to the send id of the triggering <send> element. Otherwise it *must* leave it blank.
- *origin*. This is a URI, equivalent to the 'target' attribute on the <send> element. For external events, the SCXML Processor *should* set this field to a value which, when used as the value of 'target', will allow the receiver of the event to <send> a response back to the originating entity via the Event I/O Processor specified in 'origintype'. For internal and platform events, the Processor *must* leave this field blank.
- *origintype*. This is equivalent to the 'type' field on the <send> element. For external events, the SCXML Processor *should* set this field to a value which, when used as the value of 'type', will allow the receiver of the event to <send> a response back to the originating entity at the URI specified by 'origin'. For internal and platform events, the Processor *must* leave this field blank.
- *invokeid*. If this event is generated from an invoked child process, the SCXML Processor *must* set this field to the invoke id of the invocation that triggered the child process. Otherwise it *must* leave it blank.
- *data*. This field contains whatever data the sending entity chose to include in this event. The receiving SCXML Processor *should* reformat this data to match its data model, but *must not* otherwise modify it. If the conversion is not possible, the Processor *must* leave the field blank and *must* place an error 'error.execution' in the internal event queue.

6 External Communications

6.1 Introduction

[This section is informative.]

The External Communications capability allows an SCXML session to send and receive events from external entities, and to invoke external services. *6.2 <send>* provides "fire and forget" capability to deliver events and data to any destination, including other SCXML sessions. The 'delay' attribute allows for deferred event delivery and can be used to implement a timer. The details of event transport as well as the format of the event and data are determined by the Event I/O Processor selected. Each implementation will support one or more such processor, and the author of the SCXML markup can choose the one that is appropriate for the type of endpoint he is trying to reach.

6.4 <invoke> offers a more tightly coupled form of communication, specifically the ability to trigger a platform-defined service and pass data to it. It and its child <finalize> are useful in states that model the behavior of an external service. The <invoke> element is executed after the state's <onentry> element and causes an instance of the external service to be created. The <param> and <content> elements and the 'namelist' attribute can be used to pass data to the service. Any events that are received by the state machine from the invoked component

during the invocation are preprocessed by the <finalize> handler *before* transitions are selected. The <finalize> code is used to normalize the form of the returned data and to update the data model before the transitions' "event" and "cond" clauses are evaluated.

When parallel states invoke the same external service concurrently, separate instances of the external service will be started. They can be distinguished by ids which are associated with them. Similarly, the ids contained in the events returned from the external services can be used to determine which events are responses to which invocation. Each event that is returned will be processed only by the <finalize> in the state that invoked it, but that event is then processed like any other event that the state machine receives. The finalize code can thus be thought of as a preprocessing stage that applies before the event is added to the event queue. Note that the event will be passed to all parallel states to check for transitions.

Since an invocation will be canceled when the state machine leaves the invoking state, it does not make sense to start an invocation in a state that will be exited immediately. Therefore the <invoke> element is executed upon entry into the state, but only *after* checking for eventless transitions and transitions driven by pending internal events. If any such enabled transition is found , it is taken and the state is exited immediately, without triggering the invocation. Thus invocations are triggered only when the state machine has reached a stable configuration, i.e., one that it will be staying in while it waits for external events.

6.2 <send>

[This section is normative.]

6.2.1 Overview

<send> is used to send events and data to external systems, including external SCXML Interpreters, or to raise events in the current SCXML session.

6.2.2 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
event	false	Must not occur with 'eventexpr'. If the type is http://www.w3.org/TR/scxml/#SCXMLEventProcessor, either this attribute or 'eventexpr' must be present.	EventType.datatype	none		A string indicating the name of message being generated. See *E Schema* for details on the data type.
eventexpr	false	Must not occur with 'event'. If the type is http://www.w3.org/TR/scxml/#SCXMLEventProcessor, either this attribute or 'event' must be present.	Value expression	none		A dynamic alternative to 'event'. If this attribute is present, the SCXML Processor *must* evaluate it when the parent <send> element is evaluated and treat the result as if it had been entered as the value of 'event'.
target	false	Must not occur with 'targetexpr'	URI	none	A valid target URI	The unique identifier of the message target that the platform should send the event to. See *6.2.4 The Target of Send* for details.
targetexpr	false	Must not occur with 'target'	Value expression	none	An expression evaluating to a valid target URI	A dynamic alternative to 'target'. If this attribute is present, the SCXML Processor *must* evaluate it when the parent <send> element is evaluated and treat the result as if it had been entered as the value of 'target'.
type	false	Must not occur with 'typeexpr'	URI	none		The URI that identifies the

						transport mechanism for the message. See *6.2.5 The Type of Send* for details.
typeexpr	false	Must not occur with 'type'	value expression	none		A dynamic alternative to 'type'. If this attribute is present, the SCXML Processor *must* evaluate it when the parent <send> element is evaluated and treat the result as if it had been entered as the value of 'type'.
id	false	Must not occur with 'idlocation'.	xml:ID	none	Any valid token	A string literal to be used as the identifier for this instance of <send>. See *3.14 IDs* for details.
idlocation	false	Must not occur with 'id'.	Location expression	none	Any valid location expression	Any location expression evaluating to a data model location in which a system-generated id can be stored. See below for details.
delay	false	Must not occur with 'delayexpr' or when the attribute 'target' has the value "_internal".	Duration.datatype	None	A time designation as defined in CSS2 [CSS2] format	Indicates how long the processor should wait before dispatching the message. See *E Schema* for details on the data type.
delayexpr	false	Must not occur with 'delay' or when the attribute 'target' has the value "_internal".	Value expression	None	A value expression which returns a time designation as defined in CSS2 [CSS2] format	A dynamic alternative to 'delay'. If this attribute is present, the SCXML Processor *must* evaluate it when the parent <send> element is evaluated and treat the result as if it had been entered as the value of 'delay'.
namelist	false	Must not be specified in conjunction with <content> element.	List of location expressions	none	List of valid location expressions	A space-separated list of one or more data model locations to be included as attribute/value pairs with the message. (The name of the location is the attribute and the value stored at the location is the value.) See

						5.9.2 Location Expressions for details.

6.2.3 Children

- <param>. The SCXML Processor *must* evaluate this element when the parent <send> element is evaluated and pass the resulting data to the external service when the message is delivered. Occurs 0 or more times. See 5.7 <param> for details.
- <content>. The SCXML Processor *must* evaluate this element when the parent <send> element is evaluated and pass the resulting data to the external service when the message is delivered. Occurs 0 or 1 times. See 5.6 <content> for details.

A conformant SCXML document *must* specify exactly one of 'event', 'eventexpr' and <content>. A conformant document *must not* specify "namelist" or <param> with <content>.

The SCXML Processor *must* include all attributes and values provided by <param> or 'namelist' even if duplicates occur.

If 'idlocation' is present, the SCXML Processor *must* generate an id when the parent <send> element is evaluated and store it in this location. See 3.14 IDs for details.

If a delay is specified via 'delay' or 'delayexpr', the SCXML Processor *must* interpret the character string as a time interval. It *must* dispatch the message only when the delay interval elapses. (Note that the evaluation of the `send` tag will return immediately.) The Processor *must* evaluate all arguments to <send> when the <send> element is evaluated, and not when the message is actually dispatched. If the evaluation of <send>'s arguments produces an error, the Processor *must* discard the message without attempting to deliver it. If the SCXML session terminates before the delay interval has elapsed, the SCXML Processor *must* discard the message without attempting to deliver it.

6.2.4 The Target of Send

The target of the <send> operation specifies the destination of the event. The target is defined by either the 'target' or the 'targetexpr' attribute. In most cases, the format of the target depends on the type of the target (for example a SIP URL for SIP-INFO messages or a HTTP URL for Web Services). If the value of the 'target' or 'targetexpr' attribute is not supported or invalid, the Processor *must* place the error error.execution on the internal event queue. If it is unable to dispatch the message, the Processor *must* place the error error.communication on the internal event queue.

6.2.5 The Type of Send

The type of the <send> operation specifies the method that the SCXML processor *must* use to deliver the message to its target. A conformant SCXML document *may* use either the 'type' or the 'typeexpr' attribute to define the type. If neither the 'type' nor the 'typeexpr' is defined, the SCXML Processor *must* assume the default value of http://www.w3.org/TR/scxml/#SCXMLEventProcessor. If the SCXML Processor does not support the type that is specified, it *must* place the event error.execution on the internal event queue.

SCXML Processors *must* support the following type:

Value	Details
http://www.w3.org/TR/scxml/#SCXMLEventProcessor	Target is an SCXML session. The transport mechanism is platform-specific.

For details on the http://www.w3.org/TR/scxml/#SCXMLEventProcessor type, see C.1 SCXML Event I/O Processor.

Support for HTTP POST is optional, however Processors that support it *must* use the following value for the "type" attribute:

Value	Details
http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor	Target is a URL. Data is sent via HTTP POST

For details on the http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor type, see C.2 Basic HTTP Event I/O Processor.

Processors *may* support other types such as web-services, SIP or basic HTTP GET. When they do so, they *should* assign such types the URI of the description of the relevant Event I/O Processor. Processors *may* define short form notations as an authoring convenience (e.g., "scxml" as equivalent to http://www.w3.org/TR/scxml/#SCXMLEventProcessor).

6.2.6 Message Content

The sending SCXML Interpreter *must* not alter the content of the <send> and *must* include it in the message that it sends to the destination specified in the target attribute of <send>.

Note that the document author can specify the message content in one of two mutually exclusive ways:

- An optional 'event' attribute, combined with an optional 'namelist' attribute, combined with 0 or more <param> children. Here is an example using the 'event' and 'namelist' attributes:

```
<datamodel>
<data id="target" expr="'tel:+18005551212'"/>
<data id="content" expr="'http://www.example.com/mycontent.txt'"/>
</datamodel>
   ...
<send target="target" type="x-messaging" event="fax.SEND" namelist="content"/>
```

- A single <content> child containing inline content specifying the message body. See 5.6 <content> for details.

```
<send target="csta://csta-server.example.com/" type="x-csta">
      <content>
      <csta:MakeCall>
        <csta:callingDevice>22343</callingDevice>
        <csta:calledDirectoryNumber>18005551212</csta:calledDirectoryNumber>
      </csta:MakeCall>
      </content>
</send>
```

Note that the absence of any error events does not mean that the event was successfully delivered to its target, but only that the platform was able to dispatch the event.

6.3 <cancel>

[This section is normative.]

The <cancel> element is used to cancel a delayed <send> event. The SCXML Processor *must not* allow <cancel> to affect events that were not raised in the same session. The Processor *should* make its best attempt to cancel all delayed events with the specified id. Note, however, that it can not be guaranteed to succeed, for example if the event has already been delivered by the time the <cancel> tag executes.

6.3.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
sendid	false	Must not occur with sendidexpr.	IDREF	none	The sendid of a delayed event	The ID of the event(s) to be cancelled. If multiple delayed events have this sendid, the Processor will cancel them all.
sendidexpr	false	Must not occur with sendid.	Value Expression	none	Any expression that evaluates to the ID of a delayed event	A dynamic alternative to 'sendid'. If this attribute is present, the SCXML Processor *must* evaluate it when the parent <cancel> element is evaluated and treat the result as if it had been entered as the value of 'sendid'.

A conformant SCXML document *must* specify exactly one of sendid or sendidexpr.

6.3.2 Children

None

6.4 <invoke>

[This section is normative.]

The <invoke> element is used to create an instance of an external service.

6.4.1 Attribute Details

Name	Required	Attribute Constraints	Type	Default Value	Valid Values	Description
type	false	Must not occur with the 'typeexpr' attribute.	URI	none	http://www.w3.org/TR/scxml/, http://www.w3.org/TR/ccxml/, http://www.w3.org/TR/voicexml30/, http://www.w3.org/TR/voicexml21/ plus other platform-specific values.	A URI specifying the type of the external service. See below for details.
typeexpr	false	Must not occur with the 'type' attribute.	value expression	none	Any value expression that evaluates to a URI that would be a valid value for 'type'.	A dynamic alternative to 'type'. If this attribute is present, the SCXML Processor *must* evaluate it when the parent <invoke> element is evaluated and treat the result as if it had been entered as the value of 'type'.
src	false	Must not occur with the 'srcexpr' attribute or the <content> element.	URI	None	Any URI.	A URI to be passed to the external service. See below for details.
srcexpr	false	Must not occur with the 'src' attribute or the <content> element.	Value expression	None	Any expression evaluating to a valid URI.	A dynamic alternative to 'src'. If this attribute is present, the SCXML Processor *must* evaluate it when the parent <invoke> element is evaluated and treat the result as if it had been entered as the value of 'src'.

id	false	Must not occur with the 'idlocation' attribute.	ID	none	Any valid token	A string literal to be used as the identifier for this instance of <invoke>. See *3.14 IDs* for details.
idlocation	false	Must not occur with the 'id' attribute.	Location expression	none	Any valid location expression	Any data model expression evaluating to a data model location. See *5.9.2 Location Expressions* for details.
namelist	false	Must not occur with the <param> element.	List of location expressions	none	List of valid location expressions	A space-separated list of one or more data model locations to be passed as attribute/value pairs to the invoked service. (The name of the location is the attribute and the value stored at the location is the value.) See *6.4.4 Data Sharing* and *5.9.2 Location Expressions* for details.
autoforward	false		boolean	false	true or false	A flag indicating whether to forward events to the invoked process. See below for details.

6.4.2 Children

- <param>. Element containing data to be passed to the external service. Occurs 0 or more times. See *5.7 <param>*.
- <finalize>. Element containing executable content to massage the data returned from the invoked component. Occurs 0 or 1 times. See *6.5 <finalize>* for details.
- <content>. The SCXML Processor MUST evaluate this element when the parent <invoke> element is evaluated and pass the resulting data to the invoked service. Occurs 0 or 1 times. See *5.6 <content>* for details.

Platforms *must* support http://www.w3.org/TR/scxml/ as a value for the 'type' attribute. Platforms *may* support http://www.w3.org/TR/voicexml21/, which indicates a VoiceXML 2.1 interpreter, http://www.w3.org/TR/voicexml30/, which indicates a VoiceXML 3.x interpreter, and http://www.w3.org/TR/ccxml/, which indicates a CCXML 1.0 interpreter. Platforms *may* support additional values, but they *should* assign them the URI of a description of the relevant service. Processors *may* define short form notations as an authoring convenience (e.g., "scxml" as equivalent to http://www.w3.org/TR/scxml/).

A conformant SCXML document *may* specify either the 'id' or 'idlocation' attribute, but *must not* specify both. If the 'idlocation' attribute is present, the SCXML Processor *must* generate an id automatically when the <invoke> element is evaluated and store it in the location specified by 'idlocation'. (In the rest of this document, we will refer to this identifier as the "invokeid", regardless of whether it is specified by the author or generated by the platform). The automatically generated identifier *must* have the form *stateid.platformid*, where *stateid* is the id of the state containing this element and *platformid* is automatically generated. *platformid must* be unique within the current session.

When the <invoke> element is executed, if the evaluation of its arguments produces an error, the SCXML Processor *must* terminate the processing of the element without further action. Otherwise the Processor *must* start a new logical instance of the external service specified in 'type' or 'typeexpr', passing it the URL specified by 'src' or the data specified by <content>, "namelist" or <param>. (Note that the invoked service may ignore some or all of the data passed to it. See *6.4.3 Implementation of <invoke>* for a discussion of how the passed data is treated by an invoked SCXML service.) The service instance *may* be local or remote. In addition to the explicit arguments, the Processor *must* keep track of the unique invokeid and ensure that it is included in all events that the invoked service returns to the invoking session.

When the 'autoforward' attribute is set to true, the SCXML Processor *must* send an exact copy of every external event it receives to the invoked process. All the fields specified in *5.10.1 The Internal Structure of Events* *must* have the same values in the forwarded copy of the event. The SCXML Processor *must* forward the event at the point at which it removes it from the external event queue of the invoking session for processing.

The external service *may* return multiple events while it is processing. If there is a <finalize> handler in the instance of <invoke> that created the service that generated the event, the SCXML Processor *must* execute the code in that <finalize> handler right before it removes the event from the event queue for processing. It *must not* execute the <finalize> handler in any other instance of <invoke>. Once the external service has finished processing it *must* return a special event 'done.invoke.*id*' to the external event queue of the invoking process, where *id* is the invokeid for the corresponding <invoke> element. The external service *must not* generate any other events after this done event. If the invoking session takes a transition out of the state containing the <invoke> before it receives the 'done.invoke.*id*' event, the SCXML Processor *must* automatically cancel the invoked component and stop its processing. The cancel operation *must* act as if it were the final <onexit> handler in the invoking state.

Invoked services of type http://www.w3.org/TR/scxml/, http://www.w3.org/TR/ccxml/, http://www.w3.org/TR/voicexml30/, or http://www.w3.org/TR/voicexml21/ *must* interpret values specified by the <content> element or 'src' attribute as markup to be executed. Similarly, they *must* interpret values specified by <param> element or 'namelist' attribute as values that are to be injected into their data models. For targets of other invoked service types, the interpretation of <param> and <content> elements and the 'src' and 'namelist' attributes is platform-specific. However, these services *must* treat values specified by <param> and namelist identically. They *must* also treat values specified by 'src' and <content> identically.

6.4.3 Implementation of <invoke>

The implementation of <invoke>, including communication between parent and child processes, is platform-specific, but the following requirements hold in the case where the invoked process is itself an SCXML session:

- If the 'name' of a <param> element in the <invoke> matches the 'id' of a <data> element in the top-level data declarations of the invoked session, the SCXML Processor *must* use the value of the <param> element as the initial value of the corresponding <data> element. (The top-level data declarations are those that are contained in the <datamodel> element that is a child of <scxml>.) (Note that this means that any value specified in the <data> element is ignored.) The behavior of 'namelist' is similar. If the value of a key in the namelist matches the 'id' of a <data> element in the top-level data model of the invoked session, the SCXML Processor *must* use the value of the key as the initial value of the corresponding <data> element. If the names do not match, the Processor *must not* add the value of the <param> element or namelist key/value pair to the invoked session's data model. However the Processor *may* make the values available by some other platform-specific means.
- When the invoked state machine reaches a top-level final state, the Processor *must* place the event done.invoke.*id* on the external event queue of the invoking machine, where *id* is the invokeid for this invocation. Note that reaching a top level final state corresponds to normal termination of the machine and that it cannot generate or process any further events once it is in this state.
- As described above, if the invoking state machine exits the state containing the invocation before it receives the done.invoke.*id* event, it cancels the invoked session. The method for doing this is platform-specific. However, when it is cancelled, the invoked session *must* exit at the end of the next microstep. The Processor *must* execute the <onexit> handlers for all active states in the invoked session, but it *must not* generate the done.invoke.*id* event. Once it cancels the invoked session, the Processor *must* ignore any events it receives from that session. In particular it *must not* not insert them into the external event queue of the invoking session.
- The SCXML Processor *must* support the use of SCXML Event/IO processor (*C.1 SCXML Event I/O Processor*) to communicate between the invoking and the invoked sessions. The Processor *may* support the use of other Event/IO processors to communicate between the invoking and the invoked sessions.

6.4.4 Data Sharing

[This section is informative.]

The invoked external resource is logically separate from the state machine that invokes it and does not share data with it unless the author explicitly requests this with the <param> or <content> elements and/or the 'src' and 'namelist' attributes.

The invoked and invoking process can also communicate via events. In addition to automatic forwarding specified by the 'autoforward' attribute. SCXML scripts can also use the <send> tag to send messages to the child process on an ad-hoc basis. The 'type' attribute of <send> is set to the same value as was used in the original <invoke>, while the target has the special form *#_invokeid*, where *invokeid* is the identifier corresponding to the original <invoke> tag. For example, in a document using ECMAScript as the data model, the following code would invoke a VoiceXML session:

```
<invoke type="http://www.w3.org/TR/voicexml21/" idlocation="myInvoke"/>
```

In this case, the unique invoke identifier has been stored in the data model location MyInvoke. Since the target attribute is an expression which is evaluated, the following code will extract that identifier and send a message to the invoked VoiceXML session:

```
<send type="http://www.w3.org/TR/voicexml21/" targetexpr="'#_' + myInvoke"/>
```

Finally, in the case where the invoked external service is an SCXML session, it can use <send> with the special target '_parent' and type 'scxml' to send events, possibly containing data, to the invoking session.

6.5 <finalize>

[This section is normative.]

The <finalize> element enables an invoking session to update its data model with data contained in events returned by the invoked session. <finalize> contains executable content that is executed whenever the external service returns an event after the <invoke> has been executed. This content is applied before the system looks for transitions that match the event. Within the executable content, the system variable '_event' can be used to refer to the data contained in the event which is being processed.In the case of parallel states, only the finalize code in the original invoking state is executed.

6.5.1 Attribute Details

None.

6.5.2 Children

<finalize>'s children consist of 0 or more elements of executable content.

In a conformant SCXML document, the executable content inside <finalize> *must not* raise events or invoke external actions. In particular, the <send> and <raise> elements *must not* occur.

If one or more elements of executable content is specified, the SCXML Processor *must* execute them each time an event is received from the child process that was created by the parent <invoke> element. The Processor *must* execute them right before the event is pulled off the external event queue for processing. The Processor *must not* execute them at any other time or in response to any other events.

If no executable content is specified, the SCXML Processor *must* update the data model each time an event is received from the child process that was created by the parent <invoke> element. Specifically if the parent <invoke> element contains a 'namelist' attribute or one or more <param> children containing 'location' attributes, then for each item in the 'namelist' attribute and each such <param> element, the Processor *must* update the corresponding location as if by <assign> with any return value that has a name that matches the 'namelist' item or the 'name' of the <param> element. Thus the effect of an <invoke> with an empty <finalize> element and either a 'namelist' attribute or a <param> element with a 'location' attribute is first to send the part of the data model specified by 'namelist' or 'location' to the invoked component and then to update that part of the data model with any returned values that have the same name. Note that the automatic update does not take place if the <finalize> element is absent as opposed to empty.

In the example below, a state machine using an ECMAScript data model invokes a clock object that returns the current time in a ping event with an XML payload that includes the currentSecond, currentMinute, currentHour (1-12), and an isAm flag. <finalize> maps this data into an

ECMAScript date object that is used in the condition of a transition. Thus <finalize> normalizes the data before the conditions on transitions are evaluated.

```
<scxml version="1.0" datamodel="ecmascript">
....
 <state id="getTime">
   <transition event="ping" cond="time.getHours() > 17 || time.getHours() < 9" target="storeClosed"/>
   <transition event="ping" target="takeOrder"/>
   <datamodel>
     <data id="time" expr="new Date()"/>
   </datamodel>
   <invoke id="timer" type="x-clock" src="clock.pl">
     <finalize>
       <script>
         time.setSeconds(_event.data.currentSecond);
         time.setMinutes(_event.data.currentMinute);
         time.setHours(_event.data.currentHour + (_event.isAm ? 0 : 12) - 1);
       </script>
     </finalize>
   </invoke>
 </state>
....
```

A Conformance

[This section is normative.]

A.1 Conforming Documents

The following conformance requirements hold for all SCXML documents.

1. The root element of the document *must* be <scxml>.
2. The <scxml> element *must* include a "version" attribute with the value "1.0".
3. The <scxml> element *must* designate the SCXML namespace. This can be achieved by declaring an "xmlns" attribute or an attribute with an "xmlns" prefix [XMLNames]. The namespace for SCXML is defined to be http://www.w3.org/2005/07/scxml.
4. The document *must* conform to all the syntactic constraints defined in this specification, including those contained in the schema, those contained in the "Attribute Constraints" and "Valid Values" fields in the attribute tables, and those contained in the definition of its children.
5. The document *must* conform to any additional syntactic constraints that are defined for the data model that it has chosen. See *B Data Models* for the definition of the individual data models.

A.2 Conforming Processors

A SCXML 1.0 processor is a user agent that can parse and process Conforming SCXML 1.0 documents.

In a Conforming SCXML 1.0 Processor, the XML parser *must* be able to parse and process all well-formed XML constructs defined within [XML] and [XMLNames]. It is not required that a Conforming SCXML 1.0 processor use a validating parser.

A Conforming SCXML 1.0 Processor *must* support the syntax and semantics of all mandatory SCXML elements described in this document. A Conforming SCXML 1.0 Processor *may* support the syntax and semantics of any optional SCXML elements described in this document.

When a Conforming SCXML 1.0 Processor encounters a non-conformant document, or one containing non-SCXML elements or attributes which are proprietary, or defined in a non-SCXML namespace, its behavior is undefined.

There is no conformance requirement with respect to performance characteristics of the SCXML 1.0 Processor.

B Data Models

[This section is normative.]

The 'datamodel' attribute on <scxml> defines the data model that the document uses. The data model includes the underlying data structure plus languages for boolean expressions, location expressions, value expressions, and scripting. A conformant SCXML document *may* specify the data model it uses. Conformant SCXML processors *must* support the null data model, and *may* support other data models, including the ECMAScript and XPath data models. The ECMAScript and XPath model definitions given here are normative in the sense that they define how implementations that support one of these languages *must* behave. The intent is to ensure interoperability among all processors that support ECMAScript, and all those that support XPath, without requiring all implementations to support either of those data model languages.

The definition of a data model *must*:

- Specify the boolean expression language used as the value of the 'cond' attribute in <transition>, <if> and <elseif> This language *must* not have side effects and *must* include the predicate 'In', which takes a single argument, the id of a state in the enclosing state machine, and returns 'true' if the state machine is in that state.
- Specify the location expression language that is used as the value of the 'location' attribute of the <assign> tag.
- Specify the value expression language that is used as the value of the 'expr' attribute of the <data> and <assign> elements.
- Specify the scripting language used inside the <script> element

B.1 The Null Data Model

The value "null" for the 'datamodel' attribute results in an absent or empty data model. In particular:

B.1.1 Data Model

There is no underlying data model.

B.1.2 Conditional Expressions

The boolean expression language consists of the In predicate *only*. It has the form 'In(*id*)', where *id* is the id of a state in the enclosing state machine. The predicate *must* return 'true' if and only if that state is in the current state configuration.

B.1.3 Location Expressions

There is no location expression language.

B.1.4 Value Expressions

There is no value expression language.

B.1.5 Scripting

There is no scripting language.

B.1.6 System Variables

System variables are not accessible.

B.1.7 Unsupported Elements

The <foreach> element and the elements defined in *5 Data Model and Data Manipulation* are not supported in the Null Data Model.

B.2 The ECMAScript Data Model

The value 'ecmascript' for the 'datamodel' attribute results in an ECMAScript data model. Implementations that support this value *must* support the third edition of ECMAScript [ECMASCRIPT-262]. Implementations *may* support JSON [RFC 4627] or ECMAScript for XML (E4X) [E4X].

B.2.1 Data Model

For each <data> element in the document, the SCXML Processor *must* create an ECMAScript variable object whose name is the value of the 'id' attribute of `<data>`. In cases where the 'src' attribute or in-line content is provided in the <data> element, then if an indication of the type of the content is available (e.g., via a Content-Type header), then the Processor *should* try to interpret the content according to that indication. Otherwise if the content (whether fetched or provided in-line) is JSON (and the Processor supports JSON), the SCXML Processor *must* create the corresponding ECMAScript object. Otherwise, if the content is a valid XML document, the Processor *must* create the corresponding DOM structure and assign it as the value of the variable. Otherwise the Processor *must* treat the content as a space-normalized string literal and assign it as the value of the variable. If no value is assigned, the SCXML Processor *must* assign the variable the default value ECMAScript undefined. Note that the assignment takes place at the time indicated by the 'binding' attribute on the <scxml> element.

Example: Datamodel <data> initialization

```
<scxml version="1.0" xmlns="http://www.w3.org/2005/07/scxml" datamodel="ecmascript">
  <datamodel>
   <data id="employees" src="http://example.com/employees.json"/>
   <data id="year" expr="2008"/>
   <data id="CEO" expr="'Mr Big'"/>
   <data id="profitable" expr="true"/>
  </datamodel>
</scxml>
```

B.2.2 Scoping

The Processor *must* place all variables in a single global ECMAScript scope. Specifically, the SCXML Processor *must* allow any data element to be accessed from any state. Ordering dependencies between <data> elements are not permitted. In the case of early binding, the SCXML Processor *must* evaluate all <data> elements at initialization time but *may* do so in any order it chooses. When late binding is selected, the SCXML Processor *must* create data model elements at initialization time but *may* do so in any order it chooses. Similarly, the processor *must* assign the specified initial values to data elements only when the state containing them is first entered, but *may* do so in any order it chooses.

B.2.3 Conditional Expressions

The Processor *must* convert ECMAScript expressions used in conditional expressions into their effective boolean value using the ToBoolean operator as described in Section 9.2 of [ECMASCRIPT-262].

The following example illustrates this usage.

Example: Use of a boolean expression to determine whether or not a transition is taken.

```
<state id="errorSwitch">
  <datamodel>
   <data id="time"/>
  </datamodel>

  <onentry>
    <assign location="time" expr="currentDateTime()"/>
  </onentry>

  <transition cond="yearFromDatetime(time) > 2009" target="newBehavior"/>

  <transition target="currentBehavior"/>
</state>
```

The SCXML processor *must* define an ECMAScript function named 'In()' that takes a stateID as its argument and returns 'true' if and only if that state is in the current state configuration, as described in *5.9.1 Conditional Expressions*. Here is an example of its use, taken from *G.3*

Microwave Example (Using parallel) below:

```
  <transition cond="In('closed')" target="cooking"/>
```

B.2.4 Location Expressions

The SCXML Processor *must* accept any ECMAScript left-hand-side expression as a location expression. The following example illustrates this usage. (Note that the example assumes that the data loaded from http://example.com/employees.json creates the necessary data structure, so that employees.employee[12].salary exists when <assign> is evaluated. If it didn't, the Processor would raise error.execution and the <assign> would have no effect.)

Example: Use of the location attribute of the assign to update the salary of an employee.

```
<state id="errorSwitch">
    <datamodel>
      <data id="employees" src="http://example.com/employees.json"/>
    </datamodel>

    <onentry>
        <assign location="employees.employee[12].salary" expr="42000"/>
    </onentry>
</state>
```

B.2.5 Value Expressions

The SCXML Processor *must* accept any ECMAScript expression as a value expression.

Example: Copying event data into the local data model for the state.

```
<state id="processEvent">
    <datamodel>
        <data id="myEvent"/>
    </datamodel>

    <onentry>
        <assign location="myEvent" expr="_event.data"/>
    </onentry>
</state>
```

B.2.6 <content>

When <content> is a child of <donedata>, the Processor *must* interpret its value as defined in *B.2.8.1 _event.data*. When <content> is a child of <send>, the interpretation of its value depends on the Event I/O Processor. When <content> is a child of <invoke>, the interpretation of its value is platform-specific.

B.2.7 <assign>

When evaluating an <assign> element in the ECMAScript data model, the SCXML Processor *must* replace the existing value at 'location' with the value produced by evaluating 'expr'. If it is unable to do so (for example, if the <assign> element attempts to assign to a read-only attribute), it *must* place the error error.execution on the internal event queue.

B.2.8 System Variables

The SCXML Processor *must* define an ECMAScript read-only variable for each system variable defined in *5.10 System Variables*. The `_sessionid` and `_name` system variables are defined as variables with ECMAScript String values. The `_event` system variable is defined as an object with properties for each of the fields defined in *5.10.1 The Internal Structure of Events*: `name`,`type`, `sendid`, `origin`, `origintype`, and `invokeid` are String values, while `data` can be of any type. In cases where this specification does not specify a value for one of these fields or states that the field is empty or has no value, the Processor *must* set the value to ECMAScript undefined. As the value of the _ioprocessors system variable the Processor *must* create an object with a named property for each Event I/O processor that it supports, where the name of the property is the same as that of the I/O processor and the value of the property is an object that represents the I/O processor. For the SCXML and BasicHTTP Event I/O processors, the Processor *must* create a "location" property under that object, assigning the access URI as its String value. For example, in systems that support the BasicHTTP Event I/O Processor, the access URI of the BasicHTTP Event I/O processor can be accessed as _ioprocessors['http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor'].location.

B.2.8.1 _event.data

_event.data is populated with content provided by an external event or by <param> or <content>. In some circumstances, the entity that generated the event may have indicated how the data is to be interpreted. In such cases, the Processor *should* try to format the data according to those indications. Otherwise, if the SCXML Processsor can interpret the content as key-value pairs, then for each unique key, the Processor *must* create a property of _event.data whose name is the name of the key-value pair and whose value is the value of the key-value pair. In the case of duplicate keys, the behavior is platform-specific. (Note that content specified by <param> or delivered as POST parameters via the BasicHTTP Event I/O Processor consists unambiguously of key-value pairs.) Otherwise (i.e., if the content does not consist of key-value pairs), if the Processor supports JSON and it can interpret the content as JSON, it *must* create the corresponding ECMAScript object(s) as the value of _event.data. Otherwise, if the Processor can interpret the content as a valid XML document, it *must* create the corresponding DOM structure and assign it as the value _event.data. Otherwise, the Processor *must* treat the content as a space-normalized string literal and assign it as the value of _event.data.

Suppose as part of executing a state machine named "myName" with a platform-assigned sessionid "12345", we are processing an event with the name "foo.bar" and the following object payload:

```
    { "answer" : 42 }
```

Then the underlying ECMAScript data model would have the following form:

Example: Illustration of system injected properties

```
{
  // The four properties below are automatically populated by the system

  "_name"      : "myName" ,
  "_sessionid" : "12345"  ,
  "_event"     : {
                   "name" : "foo.bar" ,
                   "data" : {
                              "answer"  :  42
                            }
                 } ,
  "_ioprocessors" : {
  "http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor" : {
  "location" : "http://example.com/scxml-http/12345"
}
  "http://www.w3.org/TR/scxml/#SCXMLEventProcessor" :  {
  "location" : "http://example.com/scxml-http/23456"
}
} ,

  // Rest of the application / developer-authored data model goes here
}
```

As an example, here is a sample transition that accesses the `_event` variable in that data model.

Example: Accessing system _event name in a condition

```
<state id="checkEventName">
    <transition cond="_event.name=='foo.bar'" target="nextState">
     ...
    </transition>
</state>
```

B.2.9 Serialization

In certain circumstances, e.g. when including data in events sent via the BasicHTTP Event I/O Processor, the SCXML Processor is required to serialize data from the ECMAScript data model for transmission to a remote entity. In such cases, if the Processor supports JSON, and is able to serialize the data in sufficient detail to allow its reconstruction, the Processor *must* serialize the data to JSON. Otherwise, the Processor *may* use platform-specific methods (including JSON despite the loss of information) to serialize the data. The Processor *should* provide a warning if the serialization entails loss of information or if it is unable to serialize at all.

B.2.10 Scripting

The SCXML Processor *must* accept any ECMAScript program as defined in Section 14 of [ECMASCRIPT-262] as the content of a <script> element.

B.2.11 <foreach>

In the ECMAScript data model, the SCXML Processor *must* support iteration over objects that satisfy instanceof(Array) in ECMAScript. It *may* support iteration over other types of objects. The legal values for the 'item' attribute are legal ECMAScript variable names. In cases where ECMAScript specification defines the iteration order for the object, the Processor *must* follow that ordering. (For arrays and strings, this goes from 0 to length-1.) In cases where the ECMAScript specification does not define the iteration order, the Processor *may* follow any order that it chooses as long as 1) it iterates over the object's own enumerable properties only 2) it visits each such property once and only once. Note that since the Processor is required to behave as if it has made a shallow copy, in the case of an array <foreach> assignment is equivalent to item = array_name[index] in ECMAScript. Note also that the assigned value could be undefined for a sparse array.

Example: Logging ISBN of all books in bookstore shopping cart

```
<foreach array="cart.books" item="book">
<log expr="'Cart contains book with ISBN ' + book.isbn"/>
</foreach>
```

C Event I/O Processors

[This section is normative.]

C.1 SCXML Event I/O Processor

The SCXML Event I/O Processor is intended to transport messages between SCXML sessions. This specification defines the mapping between the parameters in the sending session and the event that is raised in the receiving session, but the transport mechanism is platform-specific.

The sending and receiving SCXML Processors *must* maintain the following mappings:

1. The 'name' field of the event raised in the receiving session *must* match the value of the 'event' attribute of the <send> element in the sending session.
2. The 'origin' field of the event raised in the receiving session *must* match the value of the 'location' field inside the entry for the SCXML Event I/O Processor in the _ioprocessors system variable in the sending session.
3. The 'sendid' field of the event raised in the receiving session *must* match the sendid in the sending session, if the author of the sending session specifies either the 'id' or 'idlocation' attribute. If the author does not specify either the 'id' or 'idlocation' attribute, the 'sendid' field *must* be left empty.
4. The 'origintype' field of the event raised in the receiving session *must* have the value "scxml".
5. The 'data' field of the event raised in the receiving session *must* contain a copy of the data specified in the 'namelist' attribute or in <param> or <content> elements in the sending session. The nature of the copy operation depends on the data model in question.

However, the Processor *must* ensure that changes to the transmitted data in the receiving session do not affect the data in the sending session and vice-versa. The format of the 'data' field will depend on the data model of the receiving session. See *B Data Models* for details. If the data models in the sending and receiving sessions are different, the mapping between them is platform-specific.

When using the SCXML Event I/O Processor, SCXML Processors *must* support the following special targets for <send>:

- #_internal. If the target is the special term '#_internal', the Processor *must* add the event to the internal event queue of the sending session.
- #_scxml_*sessionid*. If the target is the special term '#_scxml_*sessionid*', where *sessionid* is the id of an SCXML session that is accessible to the Processor, the Processor *must* add the event to the external queue of that session. The set of SCXML sessions that are accessible to a given SCXML Processor is platform-dependent.
- #_parent. If the target is the special term '#_parent', the Processor *must* add the event to the external event queue of the SCXML session that invoked the sending session, if there is one. See *6.4 <invoke>* for details.
- #_*invokeid*. If the target is the special term '#_*invokeid*', where *invokeid* is the invokeid of an SCXML session that the sending session has created by <invoke>, the Processor *must* add the event to the external queue of that session. See *6.4 <invoke>* for details.

If neither the 'target' nor the 'targetexpr' attribute is specified, the SCXML Processor *must* add the event to the external event queue of the sending session.

If the sending SCXML session specifies a session that does not exist or is inaccessible, the SCXML Processor *must* place the error error.communication on the internal event queue of the sending session. If the receiving Processor cannot handle the data format contained in the message, the receiving Processor *must* place the error error.communication in internal queue of the session for which the message was intended and *must* ignore the message. The Processor *should* also notify the sending processor of the error. If no errors occur, the receiving Processor *must* convert the message into an SCXML event, using the mapping defined above, and insert the event into the appropriate queue, as defined in *6.2.4 The Target of Send*

C.1.1 _ioprocessors Value

SCXML Processors *must* maintain a 'http://www.w3.org/TR/scxml/#SCXMLEventProcessor' entry in the _ioprocessors system variable. The Processor *must* maintain a 'location' field inside this entry whose value holds an address that external entities can use to communicate with this SCXML session using the SCXML Event I/O Processor.

C.1.2 Examples

Here are some examples of SCXML messages sent between SCXML sessions. Each example shows the original <send> element and a transition handling the resulting event in the receiving SCXML session.

EXAMPLE 1: First, here is a message with an XML payload generated by <send> with a 'namelist':

```
SESSION1 : SENDING SESSION
Pattern: "event" attribute with an optional "namelist"

<datamodel>
  <data id="email" expr="'mailto:recipient@example.com'"/>
  <data id="content" expr="'http://www.example.com/mycontent.txt'"/>
  <data id="xmlcontent">
    <headers xmlns="http://www.example.com/headers">
      <cc>archive@example.com</cc>
      <subject>Example email</subject>
    </headers>
  </data>
</datamodel>

  ...

<send id="send-123"
      target="http://scxml-processors.example.com/session2"
      type="http://www.w3.org/TR/scxml/#SCXMLEventProcessor" event="email.send"
      namelist="email content xmlcontent"/>
```

Here is sample SCXML code to process that event in the receiving SCXML session. In this example <my:email> is platform-specific executable content that sends an email:

```
SESSION2 : RECEIVING SESSION
Pattern: "event" attribute with an optional "namelist"

<scxml:transition event="email.send">
  <my:email to="data('_event')/scxml:property[@name='email']"
            cc="data('_event')/scxml:property[@name='xmlcontent']/h:headers/h:cc"
            subject="data('_event')/scxml:property[@name='xmlcontent']/h:headers/h:subject"
            content="data('_event')/scxml:property[@name='content']"/>
</scxml:transition>
```

EXAMPLE 2: The next example shows <send> using inline XML content:

```
SESSION1 : SENDING SESSION
Pattern: "xmlns" attribute with explicit inline content

<send id="send-123"
      target="http://scxml-processors.example.com/session2"
      type="http://www.w3.org/TR/scxml/#SCXMLEventProcessor"
      xmlns:csta="http://www.ecma.ch/standards/ecma-323/csta">

  <content>
     <csta:MakeCall>
       <csta:callingDevice>22343</csta:callingDevice>
       <csta:calledDirectoryNumber>18005551212</csta:calledDirectoryNumber>
     </csta:MakeCall>
  </content>
```

```
</send>
```

Here is sample SCXML code to process the resulting event in the receiving SCXML session. It uses the special executable content <csta:makecall> to generate a telephone call:

```
SESSION2 : RECEIVING SESSION
Pattern: "xmlns" attribute with explicit inline content

<scxml:transition event="external.event">
  <csta:makecall callingDevice="data('_event')/csta:MakeCall/csta:callingDevice"
                 callingDirectoryNumber="data('_event')/csta:MakeCall/csta:callingDirectoryNumber"/>
</scxml:transition>
```

EXAMPLE 3: Finally, here is an example generated by <send> using both 'event' and 'namelist' attributes and using JSON content:

```
SESSION1 : SENDING SESSION
Pattern: "event" attribute with an optional "namelist"

<datamodel>
  <data id="email" expr="'mailto:recipient@example.com'"/>
  <data id="content" expr="'http://www.example.com/mycontent.txt'"/>
  <data id="jsoncontent" src="http://www.example.com/headers.json"/>
</datamodel>

  ...

<send sendid="send-123"
      target="'http://scxml-processors.example.com/session2'"
      type="http://www.w3.org/TR/scxml/#SCXMLEventProcessor" event="'email.send'"
      namelist="email content jsoncontent"/>
```

Here is sample SCXML code to process the resulting event in the receiving SCXML session. In this example, <my:email> is special executable content as in the first example.

```
SESSION2 : RECEIVING SESSION
Pattern: "event" attribute with an optional "namelist"

<scxml:transition event="email.send">
  <my:email to="_event.email"
            cc="_event.jsoncontent.headers.cc"
            subject="_event.jsoncontent.headers.subject"
            content="_event.content"/>
</scxml:transition>
```

C.2 Basic HTTP Event I/O Processor

The Basic HTTP Event I/O Processor is intended as a minimal interoperable mechanism for sending and receiving events to and from external components and SCXML 1.0 implementations. Support for the Basic HTTP Event I/O Processor is optional.

C.2.1 Receiving Events

An SCXML Processor that supports the Basic HTTP Event I/O Processor *must* accept messages at the access URI as HTTP POST requests (see [RFC 2616]). The SCXML Processor *must* validate the message it receives and then *must* build the appropriate SCXML event and *must* add it to the external event queue.

If a single instance of the parameter '_scxmleventname' is present, the SCXML Processor *must* use its value as the name of the SCXML event that it raises. If multiple instances of the parameter are present, the behavior is platform-specific. If the parameter '_scxmleventname' is not present, the SCXML Processor *must* use the name of the HTTP method that was used to deliver the message as the name of the event that it raises. The processor *must* use any message content other than '_scxmleventname' to populate _event.data. See *B Data Models* for details.

After it adds the received message to the appropriate event queue, the SCXML Processor *must* then indicate the result to the external component via a success response code 2XX. Note that this response is sent before the event is removed from the queue and processed. In the cases where the message cannot be formed into an SCXML event, the Processor *must* return an HTTP error code as defined in [RFC 2616].

C.2.2 Sending Events

An SCXML implementation can send events with the Basic HTTP Event I/O Processor using the <send> element (see *6.2 <send>*) with the type attribute set to "http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor" and the target attribute set to the access URI of the target. If neither the 'target' nor the 'targetexpr' attribute is specified, the SCXML Processor *must* add the event error.communication to the internal event queue of the sending session.

The SCXML Processor *must* attempt to deliver the message using HTTP method "POST" and with parameter values encoded by default in an application/x-www-form-urlencoded body (POST method). An SCXML Processor *may* support other encodings, and allow them to be specified in a platform-specific way.

If the 'event' parameter of <send> is defined, the SCXML Processor *must* use its value as the value of the HTTP POST parameter _scxmleventname. If the namelist attribute is defined, the SCXML Processor *must* map its variable names and values to HTTP POST parameters. If one or more <param> children are present, the SCXML Processor *must* map their names (i.e. name attributes) and values to HTTP POST parameters. If a <content> child is present, the SCXML Processor *must* use its value as the body of the message.

C.2.3 _ioprocessors Value

SCXML Processors that support the BasicHTTP Event I/O Processor *must* maintain a 'http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor' entry in the _ioprocessors system variable. The Processor *must* maintain a

'location' field inside this entry whose value holds an address that external entities can use to communicate with this SCXML session using the Basic HTTP Event I/O Processor.

D Algorithm for SCXML Interpretation

[This section is informative.]

This section contains an illustrative algorithm for the interpretation of an SCXML document. It is intended as a guide for implementers only. Implementations are free to implement SCXML interpreters in any way they choose.

Informal Semantics

The following definitions and highlevel principles and constraint are intended to provide a background to the algorithm, and to serve as a guide for the proper understanding of it.

Preliminary definitions

state
: An element of type <state>, <parallel>, or <final>.

pseudo state
: An element of type <initial> or <history>.

transition target
: A state, or an element of type <history>.

atomic state
: A state of type <state> with no child states, or a state of type <final>.

compound state
: A state of type <state> with at least one child state.

configuration
: The maximal consistent set of states (including parallel and final states) that the machine is currently in. We note that if a state s is in the configuration c, it is always the case that the parent of s (if any) is also in c. Note, however, that <scxml> is not a(n explicit) member of the configuration.

source state
: The source state of a transition is the state which contains the transition.

target state
: A target state of a transition is a state that the transition is entering. Note that a transition can have zero or more target states.

targetless transition
: A transition having zero target states.

eventless transition
: A transition lacking the 'event' attribute.

external event
: An SCXML event appearing in the external event queue. Such events are either sent by external sources or generated with the <send> element.

internal event
: An event appearing in the internal event queue. Such events are either raised automatically by the platform or generated with the <raise> or <send> elements.

microstep
: A microstep involves the processing of a single transition (or, in the case of parallel states, a single set of transitions.) A microstep may change the current configuration, update the data model and/or generate new (internal and/or external) events. This, by causality, may in turn enable additional transitions which will be handled in the next microstep in the sequence, and so on.

macrostep
: A macrostep consists of a sequence (a chain) of microsteps, at the end of which the state machine is in a stable state and ready to process an external event. Each external event causes an SCXML state machine to take exactly one macrostep. However, if the external event does not enable any transitions, no microstep will be taken, and the corresponding macrostep will be empty.

Principles and Constraints

We state here some principles and constraints, on the level of semantics, that SCXML adheres to:

Encapsulation
: An SCXML processor is a *pure event processor*. The only way to get data into an SCXML state machine is to send external events to it. The only way to get data out is to receive events from it.

Causality
: There shall be a *causal justification* of why events are (or are not) returned back to the environment, which can be traced back to the events provided by the system environment.

Determinism
: An SCXML state machine which does not invoke any external event processor must always react with the same behavior (i.e. the same sequence of output events) to a given sequence of input events (unless, of course, the state machine is explicitly programmed to exhibit an non-deterministic behavior). In particular, the availability of the <parallel> element must not introduce any non-determinism of the kind often associated with concurrency. Note that observable determinism does not necessarily hold for state machines that invoke other event processors.

Completeness
: An SCXML interpreter must always treat an SCXML document as *completely* specifying the behavior of a state machine. In particular, SCXML is designed to use priorities (based on document order) to resolve situations which other state machine frameworks would allow to remain under-specified (and thus non-deterministic, although in a different sense from the above).

Run to completion
: SCXML adheres to a run to completion semantics in the sense that an external event can only be processed when the processing of the previous external event has completed, i.e. when all microsteps (involving all triggered transitions) have been completely taken.

Termination
: A microstep always terminates. A macrostep may not. A macrostep that does not terminate may be said to consist of an infinitely long sequence of microsteps. This is currently allowed.

Algorithm

Note that the algorithm assumes a Lisp-like semantics in which the empty Set null is equivalent to boolean 'false' and all other entities are equivalent to 'true'.

Datatypes

These are the abstract datatypes that are used in the algorithm.

```
datatype List
   function head()      // Returns the head of the list
   function tail()      // Returns the tail of the list (i.e., the rest of the list once the head is removed)
   function append(l)   // Returns the list appended with l
   function filter(f)   // Returns the list of elements that satisfy the predicate f
   function some(f)     // Returns true if some element in the list satisfies the predicate f.  Returns false for an empty lis
   function every(f)    // Returns true if every element in the list satisfies the predicate f.  Returns true for an empty lis
The notation [...] is used as a list constructor, so that '[t]' denotes a list whose only member is the object t.

datatype OrderedSet
   procedure add(e)              // Adds e to the set if it is not already a member
   procedure delete(e)           // Deletes e from the set
   procedure union(s)            // Adds all members of s that are not already members of the set (s must also be an OrderedSe
   function isMember(e)          // Is e a member of set?
   function some(f)              // Returns true if some element in the set satisfies the predicate f.  Returns false for an e
   function every(f)             // Returns true if every element in the set satisfies the predicate f. Returns true for an em
   function hasIntersection(s)   // Returns true if this set and  set s have at least one member in common
   function  isEmpty()           // Is the set empty?
   procedure clear()             // Remove all elements from the set (make it empty)
   function  toList()            // Converts the set to a list that reflects the order in which elements were originally addec
                                 // In the case of sets created by intersection, the order of the first set (the one on which
                                 // In the case of sets created by union, the members of the first set (the one on which unior
                                 // while any members belonging to the second set only are placed after, retaining their order


datatype Queue
   procedure enqueue(e) // Puts e last in the queue
   function dequeue()   // Removes and returns first element in queue
   function isEmpty()   // Is the queue empty?

datatype BlockingQueue
   procedure enqueue(e) // Puts e last in the queue
   function dequeue()   // Removes and returns first element in queue, blocks if queue is empty

datatype HashTable // table[foo] returns the value associated with foo.  table[foo] = bar sets the value associated with foo t
```

Global variables

The following variables are global from the point of view of the algorithm. Their values will be set in the procedure interpret().

```
global configuration
global statesToInvoke
global datamodel
global internalQueue
global externalQueue
global historyValue
global running
global binding
```

Predicates

The following binary predicates are used for determining the order in which states are entered and exited.

```
entryOrder // Ancestors precede descendants, with document order being used to break ties
    (Note:since ancestors precede descendants, this is equivalent to document order.)
exitOrder  // Descendants precede ancestors, with reverse document order being used to break ties
    (Note: since descendants follow ancestors, this is equivalent to reverse document order.)
```

The following binary predicate is used to determine the order in which we examine transitions within a state.

```
documentOrder// The order in which the elements occurred in the original document.
```

Procedures and Functions

This section defines the procedures and functions that make up the core of the SCXML interpreter. N.B. in the code below, the keyword 'continue' has its traditional meaning in languages like C: break off the current iteration of the loop and proceed to the next iteration.

`procedure` interpret(scxml,id)

The purpose of this procedure is to initialize the interpreter and to start processing.

In order to interpret an SCXML document, first (optionally) perform [xinclude] processing and (optionally) validate the document, throwing an exception if validation fails. Then convert initial attributes to <initial> container children with transitions to the state specified by the attribute. (This step is done purely to simplify the statement of the algorithm and has no effect on the system's behavior. Such transitions will not contain any executable content). Initialize the global data structures, including the data model. If binding is set to 'early', initialize the data model. Then execute the global <script> element, if any. Finally call enterStates on the initial configuration, set the global running variable to true and start the interpreter's event loop.

```
procedure interpret(doc):
    if not valid(doc): failWithError()
    expandScxmlSource(doc)
    configuration = new OrderedSet()
    statesToInvoke = new OrderedSet()
    internalQueue = new Queue()
    externalQueue = new BlockingQueue()
    historyValue = new HashTable()
    datamodel = new Datamodel(doc)
    if doc.binding == "early":
        initializeDatamodel(datamodel, doc)
    running = true
    executeGlobalScriptElement(doc)
```

```
    enterStates([doc.initial.transition])
    mainEventLoop()
```

`procedure` **mainEventLoop()**

This loop runs until we enter a top-level final state or an external entity cancels processing. In either case 'running' will be set to false (see EnterStates, below, for termination by entering a top-level final state.)

At the top of the loop, we have either just entered the state machine, or we have just processed an external event. Each iteration through the loop consists of four main steps: 1)Complete the macrostep by repeatedly taking any internally enabled transitions, namely those that don't require an event or that are triggered by an internal event. After each such transition/microstep, check to see if we have reached a final state. 2) When there are no more internally enabled transitions available, the macrostep is done. Execute any <invoke> tags for states that we entered on the last iteration through the loop 3) If any internal events have been generated by the invokes, repeat step 1 to handle any errors raised by the <invoke> elements. 4) When the internal event queue is empty, wait for an external event and then execute any transitions that it triggers. However special preliminary processing is applied to the event if the state has executed any <invoke> elements. First, if this event was generated by an invoked process, apply <finalize> processing to it. Secondly, if any <invoke> elements have autoforwarding set, forward the event to them. These steps apply before the transitions are taken.

This event loop thus enforces run-to-completion semantics, in which the system process an external event and then takes all the 'follow-up' transitions that the processing has enabled before looking for another external event. For example, suppose that the *external* event queue contains events ext1 and ext2 and the machine is in state s1. If processing ext1 takes the machine to s2 and generates *internal* event int1, and s2 contains a transition t triggered by int1, the system is guaranteed to take t, no matter what transitions s2 or other states have that would be triggered by ext2. Note that this is true even though ext2 was already in the external event queue when int1 was generated. In effect, the algorithm treats the processing of int1 as finishing up the processing of ext1.

```
procedure mainEventLoop():
    while running:
        enabledTransitions = null
        macrostepDone = false
        # Here we handle eventless transitions and transitions
        # triggered by internal events until macrostep is complete
        while running and not macrostepDone:
            enabledTransitions = selectEventlessTransitions()
            if enabledTransitions.isEmpty():
                if internalQueue.isEmpty():
                    macrostepDone = true
                else:
                    internalEvent = internalQueue.dequeue()
                    datamodel["_event"] = internalEvent
                    enabledTransitions = selectTransitions(internalEvent)
            if not enabledTransitions.isEmpty():
                microstep(enabledTransitions.toList())
        # either we're in a final state, and we break out of the loop
        if not running:
            break
        # or we've completed a macrostep, so we start a new macrostep by waiting for an external event
        # Here we invoke whatever needs to be invoked. The implementation of 'invoke' is platform-specific
        for state in statesToInvoke.sort(entryOrder):
            for inv in state.invoke.sort(documentOrder):
                invoke(inv)
        statesToInvoke.clear()
        # Invoking may have raised internal error events and we iterate to handle them
        if not internalQueue.isEmpty():
            continue
        # A blocking wait for an external event.  Alternatively, if we have been invoked
        # our parent session also might cancel us.  The mechanism for this is platform specific,
        # but here we assume it's a special event we receive
        externalEvent = externalQueue.dequeue()
        if isCancelEvent(externalEvent):
            running = false
            continue
        datamodel["_event"] = externalEvent
        for state in configuration:
            for inv in state.invoke:
                if inv.invokeid == externalEvent.invokeid:
                    applyFinalize(inv, externalEvent)
                if inv.autoforward:
                    send(inv.id, externalEvent)
        enabledTransitions = selectTransitions(externalEvent)
        if not enabledTransitions.isEmpty():
            microstep(enabledTransitions.toList())
    # End of outer while running loop.  If we get here, we have reached a top-level final state or have been cancelled
    exitInterpreter()
```

`procedure` **exitInterpreter()**

The purpose of this procedure is to exit the current SCXML process by exiting all active states. If the machine is in a top-level final state, a Done event is generated. (Note that in this case, the final state will be the only active state.) The implementation of returnDoneEvent is platform-dependent, but if this session is the result of an <invoke> in another SCXML session, returnDoneEvent will cause the event done.invoke.<id> to be placed in the external event queue of that session, where <id> is the id generated in that session when the <invoke> was executed.

```
procedure exitInterpreter():
    statesToExit = configuration.toList().sort(exitOrder)
    for s in statesToExit:
        for content in s.onexit.sort(documentOrder):
            executeContent(content)
        for inv in s.invoke:
            cancelInvoke(inv)
        configuration.delete(s)
        if isFinalState(s) and isScxmlElement(s.parent):
            returnDoneEvent(s.donedata)
```

`function` **selectEventlessTransitions()**

This function selects all transitions that are enabled in the current configuration that do not require an event trigger. First find a transition with no 'event' attribute whose condition evaluates to `true`. If multiple matching transitions are present, take the first in document order. If none are present, search in the state's ancestors in ancestry order until one is found. As soon as such a transition is found, add it to enabledTransitions, and proceed to the next atomic state in the configuration. If no such transition is found in the state or its ancestors, proceed to the next state in the configuration. When all atomic states have been visited and transitions selected, filter the set of enabled transitions, removing any that are preempted by other transitions, then return the resulting set.

```
function selectEventlessTransitions():
    enabledTransitions = new OrderedSet()
    atomicStates = configuration.toList().filter(isAtomicState).sort(documentOrder)
    for state in atomicStates:
        loop: for s in [state].append(getProperAncestors(state, null)):
            for t in s.transition.sort(documentOrder):
                if not t.event and conditionMatch(t):
                    enabledTransitions.add(t)
                    break loop
    enabledTransitions = removeConflictingTransitions(enabledTransitions)
    return enabledTransitions
```

`function` selectTransitions(event)

The purpose of the selectTransitions()procedure is to collect the transitions that are enabled by this event in the current configuration.

Create an empty set of `enabledTransitions`. For each atomic state , find a transition whose 'event' attribute matches `event` and whose condition evaluates to `true`. If multiple matching transitions are present, take the first in document order. If none are present, search in the state's ancestors in ancestry order until one is found. As soon as such a transition is found, add it to enabledTransitions, and proceed to the next atomic state in the configuration. If no such transition is found in the state or its ancestors, proceed to the next state in the configuration. When all atomic states have been visited and transitions selected, filter out any preempted transitions and return the resulting set.

```
function selectTransitions(event):
    enabledTransitions = new OrderedSet()
    atomicStates = configuration.toList().filter(isAtomicState).sort(documentOrder)
    for state in atomicStates:
        loop: for s in [state].append(getProperAncestors(state, null)):
            for t in s.transition.sort(documentOrder):
                if t.event and nameMatch(t.event, event.name) and conditionMatch(t):
                    enabledTransitions.add(t)
                    break loop
    enabledTransitions = removeConflictingTransitions(enabledTransitions)
    return enabledTransitions
```

`function` removeConflictingTransitions(enabledTransitions)

enabledTransitions will contain multiple transitions only if a parallel state is active. In that case, we may have one transition selected for each of its children. These transitions may conflict with each other in the sense that they have incompatible target states. Loosely speaking, transitions are compatible when each one is contained within a single <state> child of the <parallel> element. Transitions that aren't contained within a single child force the state machine to leave the <parallel> ancestor (even if they reenter it later). Such transitions conflict with each other, and with transitions that remain within a single <state> child, in that they may have targets that cannot be simultaneously active. The test that transitions have non-intersecting exit sets captures this requirement. (If the intersection is null, the source and targets of the two transitions are contained in separate <state> descendants of <parallel>. If intersection is non-null, then at least one of the transitions is exiting the <parallel>). When such a conflict occurs, then if the source state of one of the transitions is a descendant of the source state of the other, we select the transition in the descendant. Otherwise we prefer the transition that was selected by the earlier state in document order and discard the other transition. Note that targetless transitions have empty exit sets and thus do not conflict with any other transitions.

We start with a list of enabledTransitions and produce a conflict-free list of filteredTransitions. For each t1 in enabledTransitions, we test it against all t2 that are already selected in filteredTransitions. If there is a conflict, then if t1's source state is a descendant of t2's source state, we prefer t1 and say that it preempts t2 (so we we make a note to remove t2 from filteredTransitions). Otherwise, we prefer t2 since it was selected in an earlier state in document order, so we say that it preempts t1. (There's no need to do anything in this case since t2 is already in filteredTransitions. Furthermore, once one transition preempts t1, there is no need to test t1 against any other transitions.) Finally, if t1 isn't preempted by any transition in filteredTransitions, remove any transitions that it preempts and add it to that list.

```
function removeConflictingTransitions(enabledTransitions):
    filteredTransitions = new OrderedSet()
    //toList sorts the transitions in the order of the states that selected them
    for t1 in enabledTransitions.toList():
        t1Preempted = false
        transitionsToRemove = new OrderedSet()
        for t2 in filteredTransitions.toList():
            if computeExitSet([t1]).hasIntersection(computeExitSet([t2])):
                if isDescendant(t1.source, t2.source):
                    transitionsToRemove.add(t2)
                else:
                    t1Preempted = true
                    break
        if not t1Preempted:
            for t3 in transitionsToRemove.toList():
                filteredTransitions.delete(t3)
            filteredTransitions.add(t1)

    return filteredTransitions
```

`procedure` microstep(enabledTransitions)

The purpose of the microstep `procedure` is to process a single set of transitions. These may have been enabled by an external event, an internal event, or by the presence or absence of certain values in the data model at the current point in time. The processing of the enabled transitions must be done in parallel ('lock step') in the sense that their source states must first be exited, then their actions must be executed, and finally their target states entered.

If a single atomic state is active, then enabledTransitions will contain only a single transition. If multiple states are active (i.e., we are in a parallel region), then there may be multiple transitions, one per active atomic state (though some states may not select a transition.) In this case, the transitions are taken in the document order of the atomic states that selected them.

```
procedure microstep(enabledTransitions):
    exitStates(enabledTransitions)
    executeTransitionContent(enabledTransitions)
    enterStates(enabledTransitions)
```

`procedure` exitStates(enabledTransitions)

Compute the set of states to exit. Then remove all the states on statesToExit from the set of states that will have invoke processing done at the start of the next macrostep. (Suppose macrostep M1 consists of microsteps m11 and m12. We may enter state s in m11 and exit it in m12. We will add s to statesToInvoke in m11, and must remove it in m12. In the subsequent macrostep M2, we will apply invoke processing to all states that were entered, and not exited, in M1.) Then convert statesToExit to a list and sort it in exitOrder.

For each state s in the list, if s has a deep history state h, set the history value of h to be the list of all atomic descendants of s that are members in the current configuration, else set its value to be the list of all immediate children of s that are members of the current configuration. Again for each state s in the list, first execute any onexit handlers, then cancel any ongoing invocations, and finally remove s from the current configuration.

```
procedure exitStates(enabledTransitions):
    statesToExit = computeExitSet(enabledTransitions)
    for s in statesToExit:
        statesToInvoke.delete(s)
    statesToExit = statesToExit.toList().sort(exitOrder)
    for s in statesToExit:
        for h in s.history:
            if h.type == "deep":
                f = lambda s0: isAtomicState(s0) and isDescendant(s0,s)
            else:
                f = lambda s0: s0.parent == s
            historyValue[h.id] = configuration.toList().filter(f)
    for s in statesToExit:
        for content in s.onexit.sort(documentOrder):
            executeContent(content)
        for inv in s.invoke:
            cancelInvoke(inv)
        configuration.delete(s)
```

`procedure` computeExitSet(enabledTransitions)

For each transition t in enabledTransitions, if t is targetless then do nothing, else compute the transition's domain. (This will be the source state in the case of internal transitions) or the least common compound ancestor state of the source state and target states of t (in the case of external transitions. Add to the statesToExit set all states in the configuration that are descendants of the domain.

```
function computeExitSet(transitions)
    statesToExit = new OrderedSet
    for t in transitions:
        if t.target:
            domain = getTransitionDomain(t)
            for s in configuration:
                if isDescendant(s,domain):
                    statesToExit.add(s)
    return statesToExit
```

`procedure` executeTransitionContent(enabledTransitions)

For each transition in the list of `enabledTransitions`, execute its executable content.

```
procedure executeTransitionContent(enabledTransitions):
    for t in enabledTransitions:
        executeContent(t)
```

`procedure` enterStates(enabledTransitions)

First, compute the list of all the states that will be entered as a result of taking the transitions in enabledTransitions. Add them to statesToInvoke so that invoke processing can be done at the start of the next macrostep. Convert statesToEnter to a list and sort it in entryOrder. For each state s in the list, first add s to the current configuration. Then if we are using late binding, and this is the first time we have entered s, initialize its data model. Then execute any onentry handlers. If s's initial state is being entered by default, execute any executable content in the initial transition. If a history state in s was the target of a transition, and s has not been entered before, execute the content inside the history state's default transition. Finally, if s is a final state, generate relevant Done events. If we have reached a top-level final state, set running to false as a signal to stop processing.

```
procedure enterStates(enabledTransitions):
    statesToEnter = new OrderedSet()
    statesForDefaultEntry = new OrderedSet()
    // initialize the temporary table for default content in history states
    defaultHistoryContent = new HashTable()
    computeEntrySet(enabledTransitions, statesToEnter, statesForDefaultEntry, defaultHistoryContent)
    for s in statesToEnter.toList().sort(entryOrder):
        configuration.add(s)
        statesToInvoke.add(s)
        if binding == "late" and s.isFirstEntry:
            initializeDataModel(datamodel.s,doc.s)
            s.isFirstEntry = false
        for content in s.onentry.sort(documentOrder):
            executeContent(content)
        if statesForDefaultEntry.isMember(s):
            executeContent(s.initial.transition)
        if defaultHistoryContent[s.id]:
            executeContent(defaultHistoryContent[s.id])
        if isFinalState(s):
            if isSCXMLElement(s.parent):
                running = false
            else:
                parent = s.parent
                grandparent = parent.parent
                internalQueue.enqueue(new Event("done.state." + parent.id, s.donedata))
```

```
                if isParallelState(grandparent):
                    if getChildStates(grandparent).every(isInFinalState):
                        internalQueue.enqueue(new Event("done.state." + grandparent.id))
```

`procedure` computeEntrySet(transitions, statesToEnter, statesForDefaultEntry, defaultHistoryContent)

Compute the complete set of states that will be entered as a result of taking 'transitions'. This value will be returned in 'statesToEnter' (which is modified by this procedure). Also place in 'statesForDefaultEntry' the set of all states whose default initial states were entered. First gather up all the target states in 'transitions'. Then add them and, for all that are not atomic states, add all of their (default) descendants until we reach one or more atomic states. Then add any ancestors that will be entered within the domain of the transition. (Ancestors outside of the domain of the transition will not have been exited.)

```
procedure computeEntrySet(transitions, statesToEnter, statesForDefaultEntry, defaultHistoryContent)
    for t in transitions:
        for s in t.target:
            addDescendantStatesToEnter(s,statesToEnter,statesForDefaultEntry, defaultHistoryContent)
        ancestor = getTransitionDomain(t)
        for s in getEffectiveTargetStates(t)):
            addAncestorStatesToEnter(s, ancestor, statesToEnter, statesForDefaultEntry, defaultHistoryContent)
```

`procedure` addDescendantStatesToEnter(state,statesToEnter,statesForDefaultEntry, defaultHistoryContent)

The purpose of this procedure is to add to statesToEnter 'state' and any of its descendants that the state machine will end up entering when it enters 'state'. (N.B. If 'state' is a history pseudo-state, we dereference it and add the history value instead.) Note that this procedure permanently modifies both statesToEnter and statesForDefaultEntry.

First, If state is a history state then add either the history values associated with state or state's default target to statesToEnter. Then (since the history value may not be an immediate descendant of 'state's parent) add any ancestors between the history value and state's parent. Else (if state is not a history state), add state to statesToEnter. Then if state is a compound state, add state to statesForDefaultEntry and recursively call addStatesToEnter on its default initial state(s). Then, since the default initial states may not be children of 'state', add any ancestors between the default initial states and 'state'. Otherwise, if state is a parallel state, recursively call addStatesToEnter on any of its child states that don't already have a descendant on statesToEnter.

```
procedure addDescendantStatesToEnter(state,statesToEnter,statesForDefaultEntry, defaultHistoryContent):
    if isHistoryState(state):
        if historyValue[state.id]:
            for s in historyValue[state.id]:
                addDescendantStatesToEnter(s,statesToEnter,statesForDefaultEntry, defaultHistoryContent)
            for s in historyValue[state.id]:
                addAncestorStatesToEnter(s, state.parent, statesToEnter, statesForDefaultEntry, defaultHistoryContent)
        else:
            defaultHistoryContent[state.parent.id] = state.transition.content
            for s in state.transition.target:
                addDescendantStatesToEnter(s,statesToEnter,statesForDefaultEntry, defaultHistoryContent)
            for s in state.transition.target:
                addAncestorStatesToEnter(s, state.parent, statesToEnter, statesForDefaultEntry, defaultHistoryContent)
    else:
        statesToEnter.add(state)
        if isCompoundState(state):
            statesForDefaultEntry.add(state)
            for s in state.initial.transition.target:
                addDescendantStatesToEnter(s,statesToEnter,statesForDefaultEntry, defaultHistoryContent)
            for s in state.initial.transition.target:
                addAncestorStatesToEnter(s, state, statesToEnter, statesForDefaultEntry, defaultHistoryContent)
        else:
            if isParallelState(state):
                for child in getChildStates(state):
                    if not statesToEnter.some(lambda s: isDescendant(s,child)):
                        addDescendantStatesToEnter(child,statesToEnter,statesForDefaultEntry, defaultHistoryContent)
```

`procedure` addAncestorStatesToEnter(state, ancestor, statesToEnter, statesForDefaultEntry, defaultHistoryContent)

Add to statesToEnter any ancestors of 'state' up to, but not including, 'ancestor' that must be entered in order to enter 'state'. If any of these ancestor states is a parallel state, we must fill in its descendants as well.

```
procedure addAncestorStatesToEnter(state, ancestor, statesToEnter, statesForDefaultEntry, defaultHistoryContent)
    for anc in getProperAncestors(state,ancestor):
        statesToEnter.add(anc)
        if isParallelState(anc):
            for child in getChildStates(anc):
                if not statesToEnter.some(lambda s: isDescendant(s,child)):
                    addDescendantStatesToEnter(child,statesToEnter,statesForDefaultEntry, defaultHistoryContent)
```

`procedure` isInFinalState(s)

Return true if s is a compound <state> and one of its children is an active <final> state (i.e. is a member of the current configuration), or if s is a <parallel> state and isInFinalState is true of all its children.

```
function isInFinalState(s):
    if isCompoundState(s):
        return getChildStates(s).some(lambda s: isFinalState(s) and configuration.isMember(s))
    elif isParallelState(s):
        return getChildStates(s).every(isInFinalState)
    else:
        return false
```

`function` getTransitionDomain(transition)

Return the compound state such that 1) all states that are exited or entered as a result of taking 'transition' are descendants of it 2) no descendant of it has this property.

```
function getTransitionDomain(t)
    tstates = getEffectiveTargetStates(t)
```

```
    if not tstates:
        return null
    elif t.type == "internal" and isCompoundState(t.source) and tstates.every(lambda s: isDescendant(s,t.source)):
        return t.source
    else:
        return findLCCA([t.source].append(tstates))
```

`function` findLCCA(stateList)

The Least Common Compound Ancestor is the <state> or <scxml> element s such that s is a proper ancestor of all states on stateList and no descendant of s has this property. Note that there is guaranteed to be such an element since the <scxml> wrapper element is a common ancestor of all states. Note also that since we are speaking of proper ancestor (parent or parent of a parent, etc.) the LCCA is never a member of stateList.

```
function findLCCA(stateList):
    for anc in getProperAncestors(stateList.head(),null).filter(isCompoundStateOrScxmlElement):
        if stateList.tail().every(lambda s: isDescendant(s,anc)):
            return anc
```

`function` getEffectiveTargetStates(transition)

Returns the states that will be the target when 'transition' is taken, dereferencing any history states.

```
function getEffectiveTargetStates(transition)
    targets = new OrderedSet()
    for s in transition.target
        if isHistoryState(s):
            if historyValue[s.id]:
                targets.union(historyValue[s.id])
            else:
                targets.union(getEffectiveTargetStates(s.transition))
        else:
            targets.add(s)
    return targets
```

`function` getProperAncestors(state1, state2)

If state2 is null, returns the set of all ancestors of state1 in ancestry order (state1's parent followed by the parent's parent, etc. up to an including the <scxml> element). If state2 is non-null, returns in ancestry order the set of all ancestors of state1, up to but not including state2. (A "proper ancestor" of a state is its parent, or the parent's parent, or the parent's parent's parent, etc.))If state2 is state1's parent, or equal to state1, or a descendant of state1, this returns the empty set.

`function` isDescendant(state1, state2)

Returns 'true' if state1 is a descendant of state2 (a child, or a child of a child, or a child of a child of a child, etc.) Otherwise returns 'false'.

`function` getChildStates(state1)

Returns a list containing all <state>, <final>, and <parallel> children of state1.

E Schema

[This section is informative.]

Schemas for SCXML can be found in www.w3.org/2011/04/SCXML. Two sets of schemas are available. One uses Schema 1.0 and is relatively loose, in the sense that it does not enforce all the restrictions contained in this specification. Its master schema is http://www.w3.org/2011/04/SCXML/scxml.xsd. The other set of schemas uses Schema 1.1, in particular the <assert> element, and is stricter. Its master schema is http://www.w3.org/2011/04/SCXML/scxml-strict.xsd.

F Related Work

[This section is informative.]

A number of other XML-based state machine notations have been developed, but none serves the same purpose as SCXML. XMI [UML XMI] is a notation developed for representing UML diagrams, including Harel statecharts. However it is intended as a machine interchange format and is not readily authorable by humans. ebXML [ebXML] is a language for business process specification intended to support B2B e-commerce applications. It contains a state machine language that is in some ways similar to the one presented here, but its syntax and semantics are closely tied to its intended use in e-commerce. It is therefore not suitable as a general-purpose state machine language. XTND [XTND], also called XML Transition Network Definition, is a notation for simple finite state machines but lacks Harel's notions of hierarchical and parallel states and are thus not suitable for a general-purpose state machine that is semantically equivalent to Harel state charts.

G Examples

[This section is informative.]

G.1 Language Overview

This SCXML document gives an overview of the SCXML language and shows the use of its state machine transition flows:

Example: Main.scxml

```
<?xml version="1.0" encoding="us-ascii"?>
<!-- A wrapper state that contains all other states in this file
- it represents the complete state machine -->
```

```
<scxml xmlns="http://www.w3.org/2005/07/scxml"
       xmlns:xi="http://www.w3.org/2001/XInclude"
       version="1.0"
       initial="Main"
       datamodel="ecmascript">
  <state id="Main">
    <!-- its initial state is Test1 -->
    <initial>
      <transition target="Test1"/>
    </initial>

    <!-- Really simple state showing the basic syntax. -->
    <state id="Test1">
      <initial>
        <transition target="Test1Sub1"/>
      </initial>
      <!-- Runs before we go into the substate -->
      <onentry>
        <log expr="'Inside Test1'"/>
      </onentry>

      <!-- Here is our first substate -->
      <state id="Test1Sub1">
        <onentry>
          <log expr="'Inside Test1Sub1.'"/>
        </onentry>
        <onexit>
          <log expr="'Leaving Test1Sub1'"/>
        </onexit>
        <!-- Go to Sub2 on Event1 -->
        <transition event="Event1" target="Test1Sub2"/>
      </state>

      <!-- Here is the second substate
           It is final, so Test1 is done when we get here -->
      <final id="Test1Sub2"/>

      <!-- We get this event when we reach Test1Sub2. -->
      <transition event="Test1.done" target="Test2"/>

      <!-- We run this on the way out of Test1 -->
      <onexit>
        <log expr="'Leaving Test1...'"/>
      </onexit>
    </state>

    <state id="Test2" xmlns:xi="http://www.w3.org/2001/XInclude">
      <initial>
        <transition target="Test2Sub1"/>
      </initial>

      <!-- This time we reference a state
           defined in an external file.   -->
       <xi:include href="SCXMLExamples/Test2Sub1.xml" parse="text"/>

      <final id="Test2Sub2"/>

      <!-- Test2Sub2 is defined as final, so this
           event is generated when we reach it -->
      <transition event="done.state.Test2" next="Test3"/>
    </state>

    <state id="Test3">
      <initial>
        <transition target="Test3Sub1"/>
      </initial>

      <state id="Test3Sub1">
        <onentry>
          <log expr="'Inside Test3Sub1...'"/>
          <!-- Send our self an event in 5s -->
          <send event="Timer"  delay="5s"/>
        </onentry>
        <!-- Transition on to Test4.
             This will exit both us and our parent. -->
        <transition event="Timer" target="Test4"/>
        <onexit>
          <log expr="'Leaving Test3Sub1...'"/>
        </onexit>
      </state>

      <onexit>
        <log expr="'Leaving Test3...'"/>
      </onexit>
    </state>

    <state id="Test4">
      <onentry>
        <log expr="'Inside Test4...'"/>
      </onentry>
      <initial>
        <transition target="Test4Sub1"/>
      </initial>

      <state id="Test4Sub1">
        <onexit>
          <log expr="'Leaving Test4Sub1...'"/>
        </onexit>
        <!-- This transition causes the state to exit immediately
             after entering Test4Sub1.  The transition has no event
             or guard so it is always active -->
        <transition target="Test5"/>
      </state>
    </state>
```

```
    <state id="Test5">
      <onentry>
        <log expr="'Inside Test5...'"/>
      </onentry>
      <initial>
        <transition target="Test5P"/>
      </initial>

      <!-- Fire off parallel states.  In a more realistic example
      the parallel substates Test5PSub1 and Test5PSub2 would themselves
      have substates and would do some real work before transitioning to final substates -->
      <parallel id="Test5P">
        <state id="Test5PSub1" initial="Test5PSub1Final">
           <final id="Test5PSub1Final"/>
           </state>
        <state id="Test5PSub2" initial="Test5PSub2Final">
            <final id="Test5PSub2Final"/>
            </state>
        <onexit>
          <log expr="'all parallel states done'"/>
        </onexit>
      </parallel>

      <!-- The parallel states immediately transition to final substates,
      so this event is generated immediately.   -->
      <transition event="done.state.Test5P" target="Test6"/>
    </state>

    <!--
         - This state shows invocation of an external component.
         - We will use CCXML + VoiceXML actions as an example
         - as it is a good smoke test to show how it all
         - fits together.
         - Note: In a real app you would likely
         - split this over several states but we
         - are trying to keep it simple here.
    -->
    <state id="Test6"
           xmlns:ccxml="http://www.w3.org/2002/09/ccxml"
           xmlns:v3="http://www.w3.org/2005/07/vxml3">
      <datamodel>
        <data id="ccxmlid" expr="32459"/>
        <data id="v3id" expr="17620"/>
        <data id="dest" expr="'tel:+18315552020'"/>
        <data id="src" expr="'helloworld2.vxml'"/>
        <data id="id" expr="'HelloWorld'"/>
      </datamodel>

      <onentry>
        <!-- Use <send> a message to a CCXML Processor asking it to run createcall -->
        <send target="ccxmlid" type="http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor" event="ccxml:createcall" namelist="d
      </onentry>

      <transition event="ccxml:connection.connected">
        <!-- Here as a platform-specific extension we use example V3
             Custom Action Elements instead of send. The implementation of this logic
             would be platform-dependent. -->
        <v3:form id="HelloWorld">
          <v3:block><v3:prompt>Hello World!</v3:prompt></v3:block>
        </v3:form>
      </transition>

      <transition event="v3:HelloWorld.done">
        <!-- Here we are using the low level <send>
             element to run a v3 form. Note that the event "v3:HelloWorld.done"
             is assumed either to be set/sent explicitly by the v3:form code or
             implicitly by some process outside of the v3:form -->
        <send target="v3id" type="http://www.w3.org/TR/scxml/#BasicEventProcessor" event="v3:formstart" namelist="src id"/>
      </transition>

      <transition event="v3:HelloWorld2.done">
        <!-- we use _event.data to access data in the event we're processing.
             Again we assume the v3:HelloWorld2.done is set/sent from outside
             this document -->
        <ccxml:disconnect connectionid="_event.data.connectionid"/>
      </transition>

      <transition event="ccxml:connection.disconnected" target="Done"/>

      <transition event="send.failed" target="Done">
        <!-- If we get an error event we move to the Done state that
             is a final state. -->
        <log expr="'Sending to and External component failed'"/>
      </transition>

      <onexit>
        <log expr="'Finished with external component'"/>
      </onexit>
    </state>

    <!-- This final state is an immediate child of Main
         -  when we get here, Main.done is generated. -->
    <final id="Done"/>
    <!-- End of Main > -->
  </state>
</scxml>
```

Example: Test2Sub1.xml

```
<!-- This is an example substate defined in
- an external file and included by Main.scxml.
-->
<state id="Test2Sub1">
```

```
    <onentry>
      <log expr="'Inside Test2Sub1'"/>
    </onentry>
    <transition event="Event2" target="Test2Sub2"/>
  </state>
```

G.2 Microwave Example

The example below shows the implementation of a simple microwave oven using SCXML.

Example: microwave-01.scxml

```
<?xml version="1.0"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml"
       version="1.0"
       datamodel="ecmascript"
       initial="off">

  <!--  trivial 5 second microwave oven example -->
  <datamodel>
    <data id="cook_time" expr="5"/>
    <data id="door_closed" expr="true"/>
    <data id="timer" expr="0"/>
  </datamodel>

  <state id="off">
    <!-- off state -->
    <transition event="turn.on" target="on"/>
  </state>

  <state id="on">
    <initial>
        <transition target="idle"/>
    </initial>
    <!-- on/pause state -->

    <transition event="turn.off" target="off"/>
    <transition cond="timer &gt;= cook_time" target="off"/>

    <state id="idle">
      <!-- default immediate transition if door is shut -->
      <transition cond="door_closed" target="cooking"/>
      <transition event="door.close" target="cooking">
        <assign location="door_closed" expr="true"/>
        <!-- start cooking -->
      </transition>
    </state>

    <state id="cooking">
      <transition event="door.open" target="idle">
        <assign location="door_closed" expr="false"/>
      </transition>

      <!-- a 'time' event is seen once a second -->
      <transition event="time">
        <assign location="timer" expr="timer + 1"/>
      </transition>
    </state>

  </state>

</scxml>
```

G.3 Microwave Example (Using parallel)

The example below shows the implementation of a simple microwave oven using <parallel> and the SCXML In() predicate.

Example: microwave-02.scxml

```
<?xml version="1.0"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml"
       version="1.0"
       datamodel="ecmascript"
       initial="oven">

  <!-- trivial 5 second microwave oven example -->
  <!-- using parallel and In() predicate -->
  <datamodel>
    <data id="cook_time" expr="5"/>
    <data id="door_closed" expr="true"/>
    <data id="timer" expr="0"/>
  </datamodel>

  <parallel id="oven">

    <!-- this region tracks the microwave state and timer -->
    <state id="engine">
    <initial>
      <transition target="off"/>
      </initial>

      <state id="off">
        <!-- off state -->
        <transition event="turn.on" target="on"/>
      </state>

      <state id="on">
      <initial>
        <transition target="idle"/>
```

```
        </initial>

         <!-- on/pause state -->

        <transition event="turn.off" target="off"/>
        <transition cond="timer &gt;= cook_time" target="off"/>

        <state id="idle">
          <transition cond="In('closed')" target="cooking"/>
        </state>

        <state id="cooking">
          <transition cond="In('open')" target="idle"/>

          <!-- a 'time' event is seen once a second -->
          <transition event="time">
            <assign location="timer" expr="timer + 1"/>
          </transition>
        </state>
      </state>
    </state>

    <!-- this region tracks the microwave door state -->
    <state id="door">
      <initial>
        <transition target="closed"/>
      </initial>
      <state id="closed">
        <transition event="door.open" target="open"/>
      </state>
      <state id="open">
        <transition event="door.close" target="closed"/>
       </state>
    </state>

  </parallel>

</scxml>
```

G.4 Calculator Example

The example below shows the implementation of a simple calculator in SCXML.

Example: calc.scxml

```
<?xml version="1.0" ?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
       initial="on" datamodel="ecmascript" name="calc">
    <datamodel>
         <data id="long_expr" />
         <data id="short_expr" expr="0" />
         <data id="res" />
    </datamodel>
    <state id="wrapper" initial="on">
    <state id="on" initial="ready">
        <onentry>
            <send event="DISPLAY.UPDATE" />
        </onentry>
        <state id="ready" initial="begin">
        <state id="begin">
            <transition event="OPER.MINUS" target="negated1" />
            <onentry>
                <send event="DISPLAY.UPDATE" />
            </onentry>
        </state>
        <state id="result">
        </state>
            <transition event="OPER" target="opEntered" />
            <transition event="DIGIT.0" target="zero1">
                <assign location="short_expr" expr="''" />
            </transition>
            <transition event="DIGIT" target="int1">
                <assign location="short_expr" expr="''" />
            </transition>
            <transition event="POINT" target="frac1">
                <assign location="short_expr" expr="''" />
            </transition>
        </state>
        <state id="negated1">
            <onentry>
                <assign location="short_expr" expr="'-'" />
                <send event="DISPLAY.UPDATE" />
            </onentry>
            <transition event="DIGIT.0" target="zero1" />
            <transition event="DIGIT" target="int1" />
            <transition event="POINT" target="frac1" />
        </state>
        <state id="operand1">
            <state id="zero1">
                <transition event="DIGIT" cond="_event.name != 'DIGIT.0'" target="int1" />
                <transition event="POINT" target="frac1" />
            </state>
            <state id="int1">
                <transition event="POINT" target="frac1" />
                <transition event="DIGIT">
                    <assign location="short_expr" expr="short_expr+_event.name.substr(_event.name.lastIndexOf('.')+1)" />
                    <send event="DISPLAY.UPDATE" />
                </transition>
                <onentry>
                    <assign location="short_expr" expr="short_expr+_event.name.substr(_event.name.lastIndexOf('.')+1)" />
                    <send event="DISPLAY.UPDATE" />
                </onentry>
```

```
            </state>
            <state id="frac1">
                <onentry>
                    <assign location="short_expr" expr="short_expr+'.'" />
                    <send event="DISPLAY.UPDATE" />
                </onentry>
                <transition event="DIGIT">
                    <assign location="short_expr" expr="short_expr+_event.name.substr(_event.name.lastIndexOf('.')+1)" />
                    <send event="DISPLAY.UPDATE" />
                </transition>
            </state>
            <transition event="OPER" target="opEntered" />
        </state>
        <state id="opEntered">
            <transition event="OPER.MINUS" target="negated2" />
            <transition event="POINT" target="frac2" />
            <transition event="DIGIT.0" target="zero2" />
            <transition event="DIGIT" target="int2" />
            <onentry>
                <raise event="CALC.SUB" />
                <send target="_internal" event="OP.INSERT">
                    <param name="operator" expr="_event.name" />
                </send>
            </onentry>
        </state>
        <state id="negated2">
            <onentry>
                <assign location="short_expr" expr="'-'" />
                <send event="DISPLAY.UPDATE" />
            </onentry>
            <transition event="DIGIT.0" target="zero2" />
            <transition event="DIGIT" target="int2" />
            <transition event="POINT" target="frac2" />
        </state>
        <state id="operand2">
            <state id="zero2">
                <transition event="DIGIT" cond="_event.name != 'DIGIT.0'" target="int2" />
                <transition event="POINT" target="frac2" />
            </state>
            <state id="int2">
               <transition event="DIGIT">
                    <assign location="short_expr" expr="short_expr+_event.name.substr(_event.name.lastIndexOf('.')+1)" />
                    <send event="DISPLAY.UPDATE" />
                </transition>
                <onentry>
                    <assign location="short_expr" expr="short_expr+_event.name.substr(_event.name.lastIndexOf('.')+1)" />
                    <send event="DISPLAY.UPDATE" />
                </onentry>
                <transition event="POINT" target="frac2" />
            </state>
            <state id="frac2">
                <onentry>
                    <assign location="short_expr" expr="short_expr +'.'" />
                    <send event="DISPLAY.UPDATE" />
                </onentry>
                <transition event="DIGIT">
                    <assign location="short_expr" expr="short_expr +_event.name.substr(_event.name.lastIndexOf('.')+1)" />
                    <send event="DISPLAY.UPDATE" />
                </transition>
            </state>
            <transition event="OPER" target="opEntered">
                <raise event="CALC.SUB" />
                <raise event="OP.INSERT" />
            </transition>
            <transition event="EQUALS" target="result">
                <raise event="CALC.SUB" />
                <raise event="CALC.DO" />
            </transition>
        </state>
        <transition event="C" target="on" />
    </state>
    <transition event="CALC.DO">
        <assign location="short_expr" expr="''+ res" />
        <assign location="long_expr" expr="''" />
        <assign location="res" expr="0" />
    </transition>
    <transition event="CALC.SUB">
        <if cond="short_expr!=''">
            <assign location="long_expr" expr="long_expr+'('+short_expr+')'" />
        </if>
        <assign location="res" expr="eval(long_expr)" />
        <assign location="short_expr" expr="''" />
        <send event="DISPLAY.UPDATE" />
    </transition>
    <transition event="DISPLAY.UPDATE">
        <log level="0" label="'result'" expr=".short_expr==''?res:short_expr" />
    </transition>
    <transition event="OP.INSERT">
        <log level="0" expr="_event.data[0]" />
        <if cond="_event.data[0] == 'OPER.PLUS'">
            <assign location="long_expr" expr="long_expr+'+'" />
        <elseif cond="_event.data[0]=='OPER.MINUS'" />
            <assign location="long_expr" expr="long_expr+'-'" />
        <elseif cond="_event.data[0]=='OPER.STAR'" />
            <assign location="long_expr" expr="long_expr+'*'" />
        <elseif cond="_event.data[0]=='OPER.DIV'" />
            <assign location="long_expr" expr="long_expr+'/'" />
        </if>
    </transition>
    </state>
</scxml>
```

G.5 Examples of Invoke and finalize

The following two SCXML documents demonstrate the use of Invoke and finalize. The first example shows the control flow for a voice portal offering traffic reports.

Example: Traffic Report

```
<?xml version="1.0"?>
<?access-control allow="*"?>
<scxml version="1.0" initial="Intro" datamodel="ecmascript">
  <state id="Intro">
    <invoke src="dialog.vxml#Intro" type="vxml2"/>
    <transition event="success" cond="sessionChrome.playAds" target="PlayAds"/>
    <transition event="success" cond="!sessionChrome.playAds &amp;amp;&amp;amp; ANIQuality"
                target="ShouldGoBack"/>
    <transition event="success" cond="!sessionChrome.playAds &amp;amp;&amp;amp; !ANIQuality"
                target="StartOver"/>
  </state>

  <state id="PlayAds">
    <invoke src="dialog.vxml#PlayAds" type="vxml2"/>
    <transition event="success" cond="ANIQuality" target="ShouldGoBack"/>
    <transition event="success" cond="!ANIQuality" target="StartOver"/>
  </state>

  <state id="StartOver">
    <onenter>
      <script>enterStartOver();</script>
    </onenter>
    <invoke src="dialog.vxml#StartOver" type="vxml2">
      <param name="gotItFromANI" expr="gotItFromANI"/>
      <finalize>
        <script>finalizeStartOver();</script>
      </finalize>
    </invoke>
    <transition event="success" target="ShouldGoBack"/>
    <transition event="doOver" target="StartOver"/>
    <transition event="restart" target="Intro"/> <!-- bail out to caller -->
  </state>

  <state id="ShouldGoBack">
    <invoke src="dialog.vxml#ShouldGoBack" type="vxml2">
      <param name="cityState" expr="cityState"/>
      <param name="gotItFromANI" expr="gotItFromANI"/>
      <finalize>
        <script>finalizeShouldGoBack();</script>
      </finalize>
    </invoke>
    <transition event="highWay" target="HighwayReport"/>
    <transition event="go_back" target="StartOver"/>
    <transition event="doOver" target="ShouldGoBack"/>
    <transition event="restart" target="Intro"/>
  </state>

  <state id="HighwayReport">
    <invoke src="dialog.vxml#HighwayReport" type="vxml2">
      <param name="cityState" expr="cityState"/>
      <param name="gotItFromANI" expr="gotItFromANI"/>
      <param name="playHRPrompt" expr="playHRPrompt"/>
      <param name="metroArea" expr="metroArea"/>
      <finalize>
        <script>finalizeHighwayReport();</script>
      </finalize>
    </invoke>
    <transition event="highway" target="PlayHighway"/>
    <transition event="go_back" target="StartOver"/>
    <transition event="doOver" target="HighwayReport"/>
    <transition event="fullreport" target="FullReport"/>
    <transition event="restart" target="Intro"/>
  </state>

  <state id="FullReport">
    <invoke src="dialog.vxml#FullReport" type="vxml2">
      <param name="cityState" expr="cityState"/>
      <param name="metroArea" expr="metroArea"/>
      <finalize>
        <script>finalizeFullReport();</script>
      </finalize>
    </invoke>
    <transition event="go_back" target="HighwayReport"/>
    <transition event="new_city" target="StartOver"/>
  </state>

  <state id="PlayHighway">
    <invoke src="dialog.vxml#PlayHighway" type="vxml2">
      <param name="cityState" expr="cityState"/>
      <param name="curHighway" expr="curHighway"/>
      <finalize>
        <script>finalizePlayHighway();</script>
      </finalize>
    </invoke>
    <transition event="go_back" target="HighwayReport"/>
  </state>
</scxml>
```

The following example shows a the control flow for a blackjack game.

Example: Blackjack

```
<?xml version="1.0"?>
<?access-control allow="*"?>
<scxml version="1.0" datamodel="ecmascript" initial="master"> <state id="master">
    <initial id="init1">
```

```
      <transition target="_home"/>
    </initial>
    <transition event="new_dealer" target="NewDealer"/>
    <transition event="mumble" target="_home"/> <!-- bail out to caller -->
    <transition event="silence" target="_home"/> <!-- bail out to caller -->
    <state id="_home">
      <onenter>
        <script>
        _data = {};
        </script>
      </onenter>
      <invoke src="datamodel.v3#InitDataModel" type="vxml3">
        <finalize>
          <script>
          var n;
          for (n in event) {
              _data[n] = event[n];
          }
          </script>
        </finalize>
      </invoke>
      <transition event="success" target="Welcome"/>
    </state>

    <state id="Welcome">
      <invoke src="dialog.vxml#Welcome" type="vxml3">
        <param name="skinpath" expr="skinpath"/>
      </invoke>
      <transition event="success" target="Intro2"/>
    </state>

    <state id="Intro2">
      <invoke src="dialog.vxml#Intro2" type="vxml3">
        <param name="skinpath" expr="skinpath"/>
      </invoke>
      <transition event="success" target="EvalDeal"/>
    </state>

    <state id="EvalDeal">
      <onenter>
        <script>enterEvalDeal();</script>
      </onenter>
      <invoke src="dialog.vxml#EvalDeal" type="vxml3">
        <param name="skinpath" expr="skinpath"/>
        <param name="playercard1" expr="playercard1"/>
        <param name="playercard2" expr="playercard2"/>
        <param name="playertotal" expr="blackjack.GetTotalOf('caller').toString()"/>
        <param name="dealercardshowing" expr="dealercardshowing"/>
      </invoke>
      <transition event="success" target="AskHit"/>
    </state>

    <state id="AskHit">
      <invoke src="dialog.vxml#AskHit" type="vxml3">
        <param name="skinpath" expr="skinpath"/>
        <finalize>
          <script>finalizeAskHit();</script>
        </finalize>
      </invoke>
      <transition event="hit" target="PlayNewCard"/>
      <transition event="stand" target="PlayDone"/>
    </state>

    <state id="PlayNewCard">
      <invoke src="dialog.vxml#PlayNewCard" type="vxml3">
        <param name="skinpath" expr="skinpath"/>
        <param name="playernewcard" expr="playernewcard"/>
        <param name="playertotal" expr="blackjack.GetTotalOf('caller').toString()"/>
      </invoke>
      <transition event="success" cond="blackjack.GetTotalOf('caller') &gt;= 21" target="PlayDone"/>
      <transition event="success" target="AskHit"/> <!-- less than 21 -->
    </state>

    <state id="PlayDone">
      <onenter>
        <script>enterPlayDone();</script>
      </onenter>
      <invoke src="dialog.vxml#PlayDone" type="vxml3">
        <param name="skinpath" expr="skinpath"/>
        <param name="gameresult" expr="blackjack.GetGameResult()"/>
        <param name="dealertotal" expr="blackjack.GetTotalOf('dealer').toString()"/>
      </invoke>
      <transition event="playagain" target="Intro2"/>
      <transition event="quit" target="_home"/>
    </state>

    <state id="NewDealer">
      <onenter>
       <script>enterNewDealer();</script>
      </onenter>
      <invoke src="dialog.vxml#Dummy" type="vxml3"/>
      <transition event="success" target="Welcome"/>
    </state>
  </state>
</scxml>
```

G.6 Inline Content and Namespaces

Since SCXML documents are XML documents, normal XML namespace rules apply to inline content specified with <content> and <data>. In particular, if no namespace is specified, the inline content will be placed in the SCXML namespace. Consider the following example:

```
<send target="http://example.com/send/target" type="'http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor'">
  <content>
    <a>fffff</a>
  </content>
 </send>
```

The recipient of the message will see the following:

```
<a xmlns="http://www.w3.org/2005/07/scxml">fffff</a>
```

The following markup would cause the message to be delivered without namespaces:

```
<send target="http://example.com/send/target" type="'http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor'">
  <content>
    <a xmlns="">fffff</a>
  </content>
 </send>
```

The recipient of the message will see the following:

```
<a>fffff</a>
```

The sender can also specify multiple namespaces:

```
<send target="http://example.com/send/target" type="'http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor'">
 <content>
   <root xmlns="http://example.com/r" xmlns:a="http://example.com/a" xmlns:b="http://example.com/b" xmlns:c="http://example.co
      <a:alpha>1</a:alpha>
      <b:beta>2</b:beta>
      <c:gamma>3</c:gamma>
   </root>
 </content>
</send>
```

In this case, the receiver would see:

```
  <root xmlns="http://example.com/r">
      <alpha xmlns="http://example.com/a">1</alpha>
      <beta xmlns="http://example.com/b>2</beta>
      <gamma xmlns="http://example.com/c>3</gamma>
  </root>
```

G.7 Custom Action Elements

Custom Action Elements can be defined in other specifications/namespaces and are responsible for performing actions on behalf of custom components. Logically Custom Action Elements can be thought of as a collection of actions and handlers to perform specific tasks. An example of this is a CCXML <accept> element that is a Custom Action Element:

```
<transition event="ccxml:connection.alerting">
  <ccxml:accept connectionid="_event.data.connectionid"/>
</transition>
```

This could be written using a <send> element using the following syntax:

```
<datamodel>
<data name="connectionid"/>
</datamodel>
<transition event="ccxml:connection.alerting">
  <assign location="connectionid" expr="_event.data.connectionid"/>
 <send type="ccxml" event="ccxml:accept" namelist="connectionid"/>
</transition>
```

A more complicated example might be a CCXML <createcall> where you are both providing variables and getting values back that using only the <send> syntax would be more complex as it would need to be broken over several steps. For example:

```
<onentry>
  <ccxml:createcall dest="'tel:+18315552020'" connectionid="myConnectionID"/>
</onentry>
```

Would need to be modeled in two steps using <send> as you would need to do something like the following:

```
<datamodel>
  <data name="dest" expr="'tel:+18315552020'"/>
  <data name="connectionid"/>
```

```
  </datamodel>
  <onentry>
    <send type="ccxml" event="ccxml:createcall" namelist="dest"/>
  </onentry>
  <transition event="ccxml:createcall.success">
     <assign location="connectionid" expr="_event.data.connectionid"/>
  </transition>
```

The exact mappings between Custom Action Elements and <send> actions are to be defined in the individual Custom Action Element specifications.

H MIME Type

[This section is normative.]

This appendix registers a new MIME media type, "`application/scxml+xml`".

The "`application/scxml+xml`" media type is being submitted to the IESG for review, approval, and registration with IANA.

H.1 Registration of MIME media type application/scxml+xml

MIME media type name:
`application`
MIME subtype name:
`scxml+xml`
Required parameters:
None
Optional parameters:
`charset`
This parameter has identical semantics to the `charset` parameter of the `application/xml` media type as specified in [RFC 3023] or its successor.
Encoding considerations:
By virtue of SCXML content being XML, it has the same considerations when sent as "application/scxml+xml"as does XML. See [RFC 3023] (or its successor), section 3.2.
Security considerations:
SCXML elements may include arbitrary URIs. Therefore, the security issues of [RFC 3986] section 7 should be considered. In addition, because of the extensibility features for SCXML, it is possible that "`application/scxml+xml`" may describe content that has security implications beyond those described here. However, if the processor follows only the normative semantics of this specification, this content will be ignored. Only in the case where the processor recognizes and processes the additional content, or where further processing of that content is dispatched to other processors, would security issues potentially arise. And in that case, they would fall outside the domain of this registration document.
Interoperability considerations:
This specification describes processing semantics that dictate behavior that must be followed when dealing with, among other things, unrecognized elements.Because SCXML is extensible, conformant "`application/scxml+xml`" processors MAY expect that content received is well-formed XML, but processors SHOULD NOT assume that the content is valid SCXML or expect to recognize all of the elements and attributes in the document.
Published specification:
This media type registration is extracted from Appendix H of the State Chart XML (SCXML): State Machine Notation for Control Abstraction specification.
Additional information:
Magic number(s):
There is no single initial octet sequence that is always present in SCXML documents.
File extension(s):
SCXML documents are most often identified with the extensions "`.scxml`".
Macintosh File Type Code(s):
TEXT
Person and email address to contact for further information:
Kazuyuki Ashimura, <ashimura@w3.org>.
Intended usage:
COMMON
Restrictions on usage:
None
Author:
The SCXML specification is a work product of the World Wide Web Consortium's Voice Browser Working Group.
Change controller:
The W3C has change control over these specifications.

H.2 Fragment Identifiers

For documents labeled as "`application/scxml+xml`", the fragment identifier notation is exactly that for "`application/xml`", as specified in RFC 3023.

I References

I.1 Normative References

ECMASCRIPT-262
ECMAScript Language Specification. Standard ECMA-262, Edition 5.1 ECMA. June 2011. (See http://www.ecma-international.org/publications/standards/Ecma-262.htm.)
E4X
ECMAScript for XML (E4X) Specification Standard ECMA-357, 2nd Edition, December 2005. (See http://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/Ecma-357.pdf.)
RFC 2119
RFC 2119: Key words for use in RFCs to Indicate Requirement Levels Internet Engineering Task Force. March 1997. (See http://www.ietf.org/rfc/rfc2119.txt.)
RFC 2396

RFC 2396: Uniform Resource Identifiers Internet Engineering Task Force. August 1998. (See http://www.ietf.org/rfc/rfc2396.txt.)

RFC 2616

RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1 Internet Engineering Task Force. June 1999. (See http://www.ietf.org/rfc/rfc2616.txt.)

RFC 4627

The application/json Media Type for JavaScript Object Notation (JSON) Internet Engineering Task Force. July 2006. (See http://www.ietf.org/rfc/rfc4627.txt.)

RFC 3023

RFC 3023: XML Media Types. Internet Engineering Task Force. January 2001. (See http://www.ietf.org/rfc/rfc3023.txt.)

RFC 3986

RFC 3986: Uniform Resource Identifier (URI): Generic Syntax. Internet Engineering Task Force. January 2005. (See http://www.ietf.org/rfc/rfc3986.txt.)

XML

Extensible Markup Language (XML) 1.0 (Fifth Edition) World Wide Web Consortium. W3C Recommendation, November 2008 (See http://www.w3.org/TR/xml/.)

XMLNames

Namespaces in XML 1.0 (Second Edition) World Wide Web Consortium. W3C Recommendation, August 2006 (See http://www.w3.org/TR/2006/REC-xml-names-20060816/.)

XML Schema

XML Schema Part 2: Datatypes Second Edition World Wide Web Consortium. W3C Recommendation, October 2004. (See http://www.w3.org/TR/xmlschema-2/.)

I.2 Informative References

ebXML

ebXML Business Process Specification Schema v2.0 (See https://www.oasis-open.org/committees/documents.php?wg_abbrev=ebxml-bp.)

EL

EL: The JSP 2.0 Expression Language Interpreter (See http://commons.apache.org/el/.)

Harel and Politi

Modeling Reactive Systems with Statecharts: The STATEMATE Approach By D. Harel and M. Politi. McGraw-Hill, 1998. (See http://www.wisdom.weizmann.ac.il/~dharel/reactive_systems.html.)

CCXML 1.0

Voice Browser Call Control: CCXML Version 1.0 World Wide Web Consortium. W3C Recommendation, July 2011. (See http://www.w3.org/TR/2011/REC-ccxml-20110705/.)

VoiceXML 2.0

VoiceXML 2.0: World Wide Web Consortium. W3C Recommendation, March 2004. (See http://www.w3.org/TR/voicexml20/.)

UML 2.3

UML Specification Version 2.3 OMG, 2009. (See http://www.omg.org/spec/UML/2.3/.)

xinclude

XML Inclusions (XInclude) Version 1.0 (Second Editiion) W3C Recommendation, 2006. (See http://www.w3.org/TR/xinclude/.)

UML XMI

XML Metadata Exchange version 2.4.1 OMG. August 2011. (See http://www.omg.org/spec/XMI/.)

XTND

XML Transition Network Definition World Wide Web Consortium. W3C Note, November 2000. (See http://www.w3.org/TR/2000/NOTE-xtnd-20001121/.)