Florence XML (FXML)

Editor in chief:
Charles Lewis, AT&T
Editors:
Giuseppe Di Fabbrizio, AT&T
Michael Johnston, AT&T
Contributors:
Giovanni Andreani, SpeechVillage
Paolo Baggia, Loquendo

Version Information:
April 10, 2006 - First draft.

Document status

This is an Editors draft. It is meant for internal review only by the MMI WG Editors and a restricted number of MMI WG members. Its contents have not been endorsed or approved by the MMI WG, or by any other entity of W3C. Not for redistribution.

Table of Contents

1. Introduction
2. The Role of the IM
3. The Florence Model of Flow Control
4. The Florence Events
5. Fundamental FXML Syntax
   - 5.1 The Florence IM
     - 5.1.1 Root element: `fxml`
     - 5.1.2 Application configuration element: `configuration`
     - 5.1.3 Variable definitions: `local`, `global`
     - 5.1.4 Action definition: `actiondef`
   - 5.2 Conditions and Instructions
     - 5.2.1 Changing the value of a variable: `set`
     - 5.2.2 Checking conditions: `conditions`, `cond`, and `ucond`
     - 5.2.3 Host access instructions: `host-access`, `jdbc-access`, and `file-access`
6. Flow Controller Syntax
   - 6.1 The Recursive Transition Network (RTN) Flow Controller
     - 6.1.1 RTN top-level element: `rtn`
     - 6.1.2 RTN state: `state`
     - 6.1.3 RTN transition: `transition`
   - 6.2 The Clarification Flow Controller
     - 6.2.1 Clarification top-level element: `clarify`
     - 6.2.2 Clarification tree node: `node`

7. XPath in FXML
8. FXML application examples
  8.1 Command line "Hello, World!"
  8.2 HTML "Hello, World!"
  8.3 HTML + VXML "Hello, World!"
  8.4 "Hello, World!" with tricks
    8.4.1 Hello loop
    8.4.2 NLSML parsing
    8.4.3 HTML output
Appendices
  Appendix A. Florence XML schema
  Appendix B. References

1. Introduction

This document introduces FXML, a multimodal interaction manager markup language. It covers the FXML syntax , the role and behavior of a Florence Interaction Manager (IM) in a multimodal system, and details of the Florence model of dialogue management.

The Florence framework is a model for multimodal dialogue management. The Florence IM is an implementation of this design, and FXML is the language that is used to describe multimodal dialogue applications for this framework. The Florence design is centered around turn-based interaction with a call stack of dialogues. FXML is used to define each of these dialogues, as well as to describe global application information. DOM and XPath standards are used in FXML to maintain dialogue state information and to parse new input. ECMA script is also used as a data model to control the dialogue flow.

FXML includes markup for a dialogue model based on recursive transition networks (RTN), and can be extended to include other models. The RTN model describes dialogue states and the transitions between them, and is sufficient for most interactions. The Florence framework permits other models to be developed and used, along with or independently of the RTN model, such as rule-based or taxonomy based dialogue models.

2. The role of the IM

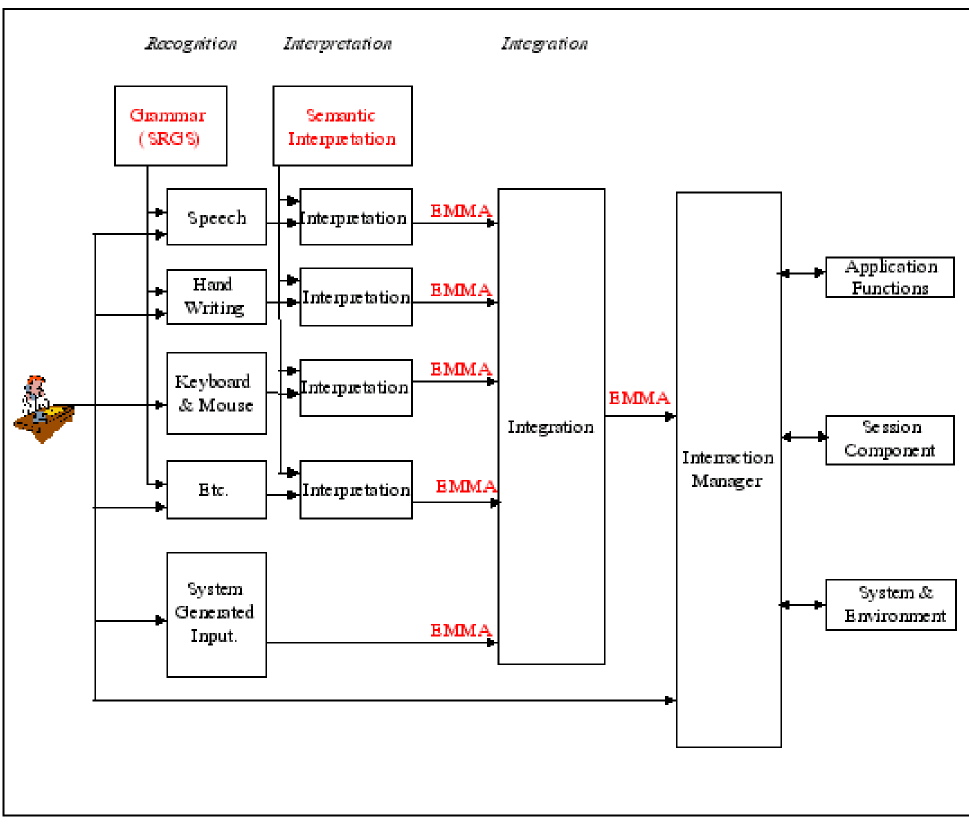
The primary responsibility of the IM is to guide, based on the input and the control flow data, the system side of a dialogue. Control flow can be described in many different ways: by a flow chart, by a rule-based system, or by any other decision-making algorithm. FXML does not mandate the use of a particular method; instead it provides a framework for applying these methods within a turn-based dialogue system. This framework, not the specific decision-making algorithm, supports the IM role.

The input to the Florence IM is most commonly in a standard data interchange XML format such as EMMA or NLSML. This input is generated externally, and can be the fusion of inputs from more than one source. In a multimodal environment, an external component handles mode synchronization and submits the integrated results to the IM. Optionally, the IM can also submit input to a second level of pre-processing that is more specific to the application. This second level of processing is more context-sensitive because of its contact with the IM. For example, if the context of a dialogue changes significantly, the IM can alert this second-level preprocessor or suggest a new grammar. The IM can also resubmit input to this second-level preprocessor when the context of the dialogue has changed enough to warrant a re-interpretation of the input. For

instance, if the preprocessor is using a simple yes/no grammar and the IM is not satisfied with the confidence level of the result, it can tell the preprocessor to change to a broader grammar and re-interpret the input.

The IM processes user input and generates the system side of the dialogue. To generate this, it uses call flow information and also has access to external data sources (e.g., database queries or any XHTML document identified by a URI web resource). The IM can use this information for decision-making or as part of the system reply.

When input processing is complete, the Florence IM uses an output template to construct a reply to the initial request. This template can be in XHTML, VXML, or any other XML output format. The IM fills in the templates based on the dialogue flow decisions, and from state and session information. Additional output can be saved to files, which is useful if we wish to generate outputs for multiple modalities.



As shown in this diagram from [5], The Multimodal Interaction Framework (MMI Framework) suggests an EMMA document for the IM input . This framework also lays out a distinct component to create the IM input document from the fusion of multiple modal component outputs. In the context of this MMI framework, the Florence IM receives this EMMA document as input and submits output to a generation component, which then responds to the user through the available modalities. This is consistent with the FXML turn-taking model.

3. The Florence model of Control Flow

The Florence model of interaction management uses a stack structure to nest dialogues. Each subdialogue maintains its own local variables (the subdialogue context), and operates by whatever decision algorithm the application author has chosen or created. Outside of this dialogue stack, the Florence IM maintains a global variable space (the context of the interaction) and information about preprocessing and output formatting.

One of the key abstractions in the Florence framework is the Flow Controller (FC). This is the object

that defines the decision logic. Each dialogue in the stack is an instance of a FC, and different FCs can be nested to allow different types of algorithms to handle different parts of the application. In object-oriented terms, the Flow Controller is an abstract class, and each implementation is a concrete subclass. The FC is strongly influenced by object-oriented concepts such as polymorphism and encapsulation.

The Recursive Transition Network (RTN) model of flow control is one example of a FC. This model is functionally similar to the flow-chart style often used by interface experts to define dialogues. An RTN FC is defined with nodes and arcs, with conditions on the arcs to control the dialogue flow and commands in both nodes and arcs. Each arc can also incrementally add to the output. Nodes (states) can be flagged as "pause" points, where output is generated and new user input is acquired.

The variables in each Flow Controller, RTN or other, as well as global variables, are accessible through a DOM structure. Structured XML can go directly into the memory space of a Florence application, and XPath can be used to query and operate on it. The global memory space is visible to every dialogue on the stack, while the local memory space is only visible to the immediate dialogue. Variable values can be passed between dialogues when a dialogue is created or completed.

FXML also supports a second global control structure (in addition to the stack-based model), which behaves something like an exception handling mechanism. It allows program execution to jump up the stack without completing intervening subdialogues. This is included to allow global conditions (such as a request for assistance, or a request to quit the application) to be handled in a uniform way throughout an application. The context shifts can be defined at a high level, and passed along to new subdialogues. This allows authors to add new FXML dialogues to an existing FXML application without compromising global behavior.

Output from the Florence IM is usually in an XML format, although this is not enforced. The output from a Florence IM is based on a template created by the application author. Segments of this template are filled in as the control flow is executed, and the filled template is returned when the call flow indicates that the IM should pause for new user input. The variable segments of the template can be incrementally added to by the call flow, such that the final output is the accumulation of multiple actions (action queuing). For example, a dialogue flow might call for an action that speaks (or prints) "Hello, World!", followed by a second action that says "Goodbye, now!" before the system pauses for new input. The output contains both phrases.

It is possible for a control flow to generate data that can be used in more than one template. This makes it simple to modify an application for new environments by changing the output template of the application. If the application generates output that cannot be included in the current template, it will be ignored. Section 8 demonstrates how to run the same application with different modalities, each with its own output template.

4. Florence Events

The Florence IM accepts (or for notification events, creates) the following events. The first 12 are for integration into the MMI Framework.

Run	Indicates that Component should begin executing
Running	Notification event returned by Florence to indicated that it has started
Halt	Indicates that Florence should cease executing

Halted	Notification event generated after receiving a halt event or after the conclusion of the dialogue application.
AddSession	Instructs the IM to prepare for a new session
SessionAdded	Notification event returned by the IM
EndSession	Indicates the end of a user session
SessionEnded	Notification event returned by the IM
Pause	Equivalent to Halt.
Paused	Notification event returned by IM
Resume	Equivalent to Run.
Resumed	Notification event returned by IM
FlorenceInput	Should include input data.
FlorenceSilence	Indicates no input.
FlorenceRejection	Indicates bad input.
FlorenceReload	Instructs the IM to reload all data.
FlorenceRestart	Instructs the IM to restart the dialogue application.
FlorenceError	Florence Runtime Error
FlorenceOutput	Generated by Florence, includes output information

The implementation of these events if completely dependent on the implementation architecture. For example, in the command line version of the IM, they must be communicated directly, while in a J2EE environment they might be part of an EJB wrapper.

5. Fundamental FXML Syntax

FXML applications have three types of data: application configuration, global variables, and flow controller definitions. These can be combined into a single file, like this:

```
<fxml version="1.4">
  <configuration>
```

```
      <output template="HelloTemplate.html"/>
    </configuration>
    <rtn name="HW-toplevel" start="start">
      <actiondefs>
        <actiondef name="helloWorld" text="Hello, World!"/>
      </actiondefs>

      <states>
     <state name="start">
          <transitions>
            <transition from="start" to="end" name="t1">
              <actions>
                <action>helloWorld</action>
              </actions>
            </transition>
          </transitions>
        </state>
        <state name="end" final="true"/>
      </states>
    </rtn>
  </fxml>
```

or organized into different files for each child element. Separation may facilitate re-use.

5.1 The Florence IM

These are the basic elements that define the IM outside of control flow code. This includes settings for debugging output and logging, as well as configuring the IM to use an external preprocessor, an output template, and context shifts.

5.1.1 Root element: `fxml`

Annotation	fxml
Definition	The root element of a FXML document.
Children	There are three possible children for an `fxml` element. At least one must be present. `configuration`: definition of the application configuration. A dialogue definition, such as `rtn`. `global`: definition of global variables

Attributes	• **Required**:
	  ○ `version`: the version of FXML that the application is written for.
	• **Optional**:
	  ○ `xmlns`
	  ○ `xmlns:xsi`
	  ○ `xsi:schemaLocation`

Applies to	None

The root element of an FXML document is named `fxml`. The `configuration` child element contains information about the configuration of the application. There can only be one `configuration` element in an FXML application.

```
<fxml version="1.4">
  <configuration>
    <dialogfile>./HelloWorld.fxml</dialogfile>
    <output template="HelloTemplate.html"/>
  </configuration>
</fxml>
```

A **global** element can also be a child of the **fxml** element. This element is used to define the global variables of an application. There is can only be one **global** element in an FXML application.

```
<fxml version="1.4">
  <global>
    <var name="userName" expr="defaultUser"/>
    <var name="dialogueExchanges" expr="0"/>
    <var name="confidenceThreshold" expr=".8"/>
  </global>
</fxml>
```

Any other child of the **fxml** element is a dialogue definition. The set of possible dialog elements is open-ended. These are elements that define the control flow of a dialogue, and there may be any number of these definitions in an application. The most common type of control flow is the Recursive Transition Network (RTN), which works like a state machine. The definition for this type of flow controller is explored in more detail elsewhere in this document, but here is a simple example:

```
<fxml>
  <rtn name="HW-toplevel" start="start">
    <actiondefs>
      <actiondef name="helloWorld" text="Hello, World!"/>
    </actiondefs>

    <states>
   <state name="start">
        <transitions>
          <transition from="start" to="end" name="t1">
            <actions>
              <action>helloWorld</action>
            </actions>
          </transition>
        </transitions>
       </state>
       <state name="end" final="true"/>
     </states>
   </rtn>
</fxml>
```

5.1.2 Application configuration element: configuration

Annotation	configuration
Definition	Defines the configuration of a dialogue application
Children	There are four possible children for a **configuration** element. Only one of them may be present. **dialogfile**: A reference to an fxml dialogue definition required for the application. **nlu** (optional): Connection information for a web service that does secondary pre-processing, called by the Florence IM **output**: Data related to output from the IM, including the location of the output

	template `logfile`: URI for saving log messages generated by the IM.
Attributes	• **All optional**: ◦ `logfile`: If the log messages are to be sent to a local file, it can be identified by name here. ◦ `debuglevel`: Granularity of debug messages generated by the IM. ◦ `consolemessages`: "true" if messages can be sent to the console. ◦ `checkvars`: "true" if the IM should check FXML code for variable declarations at load-time. ◦ `globals`: The location of an FXML file that defines global variables for this applicaiton. ◦ `afterdialog`: The location of an FXML dialog definition to be executed when the main dialogue is complete.
Applies to	None

This element is always a child of an `fxml` element. It describes the deployment specifics for an FXML application, such as settings for debugging output and logging, and where to find the external preprocessor and output template. If the `dialogfile` child is not present, the first flow controller in the document order of the `fxml` element will be used as the top-level dialogue.

For example:

```
<configuration logfile="logs.txt" consolemessages="true">
  <dialogfile>./HelloWorld.fxml</dialogfile>
  <output template="HelloTemplate.html"/>
</configuration>
```

The `configuration` element in this example tells us that the topmost dialogue in this application is defined in the "HelloWorld.fxml" file, and that the template "HelloTemplate.html" is used to generate output. The attributes of the `configuration` element indicate that the application is allowed to display console messages (not always the case for deployed apps), and that log entries are saved directly to a local file called "logs.txt".

5.1.3 Variable definitions: `<local>`, `<global>`

Annotation	**local** or **global**
Definition	Variable definitions
Children	There are three types of variables in FXML: `<var>` (optional): A variable declaration. `<array>` (optional): An array declaration. `<dictionary>` (optional): An associative array declaration.
Attributes	None
Applies to	None

The `local` and `global` elements define the variables where information about the dialogue is

maintained and communicated between dialogues, templates, actions, and other FXML elements at runtime. The `global` element is a child of the top-level `fxml` element, and can only be used once in an application. Multiple definitions of the global element will result in a FlorenceError event. The variables defined in the `global` element are visible to all dialogues in the application. The `local` element appears in a dialogue definition, and contains variables that are only visible within that dialogue. Although they differ in scope and usage, these two element types have identical syntax.

```
<global>
    <var name="userName" expr="defaultUser"/>
    <var name="dialogueExchanges" expr="0"/>
    <var name="confidenceThreshold" expr=".8"/>
</global>
```

5.1.4 Action definition: `<actiondef>`

Annotation	actiondef
Definition	Definition of a system output
Children	`fragment`: XML fragments for output
Attributes	`name`: The name of the action `text` (optional): Can be used for text output and debugging. `args` (optional): An argument list for the action. `template` (optional): The name of a file with a new template to use.
Applies to	None

The `actiondef` element defines output for the user. The definition includes a name, which is then invoked elsewhere inside of the dialogue or passed to a subdialogue. To accommodate the inclusion of mode-specific output, the `actiondef` element may have children of any type. When the action is invoked, these children (called "fragments") are plugged into the output template. If multiple actions are invoked before the output is created, the fragments are all collected and included.

For example,

```
<actiondef name="intro" text="Welcome to Florence!"/>
```

Is a simple action definition with no fragments. The basic text of the action is used when interacting directly with the IM through a command line. To this simple action, we can add fragments to describe it in different types of modalities:

```
<actiondef name="intro" text="Welcome to Florence!">
  <fragment name="HTML-div">
    <div><b>Welcome to Florence!</b></div>
  </fragment>
  <fragment name="VXML-prompt">
    <prompt>Welcome to Florence!</prompt>
  </fragment>
</actiondef>
```

In this case, we have added on fragment to be used in an HTML template where the `fragmentvar` element appears with the name "HTML-div", and another to appear in a VXML template where the `fragmentvar` element appears with the name "VXML-prompt". An action definition can include an

arbitrary number of fragments like these. Fragments that do not match to a slot in the template are discarded when output is generated, so there is no harm in over-describing an action in an FXML application. If multiple actions are called, the fragments are accumulated and all of them are added to the final output, permitting cumulative output to be generated.

The **template** element also allows an action to change the template that is being used. This change is local to the dialogue, and will revert to the application default when the dialogue returns. Sub-dialogues, however, will continue to use the new template indicated. The name of the current template is maintained as a global variable, and is available to conditions and instructions by referencing the system variable "result".

5.2 Conditions and Instructions

A "condition" is an element that contains code to produce a boolean value to determine the control flow. An "instruction" is an element that contains code to change values of local and/or global variables. The code used in both conditions and instructions may be implemented with either ECMA script or XPath. Code can also be used to describe the initial values of variables, to construct output text, and to fill attribute slots of some elements (such as for http parameters, log messages, and SQL queries). The user input is accessible to all conditions and instructions.

5.2.1 Changing the value of a variable: set

Annotation	set
Definition	An instruction for changing variable values.
Children	Optionally, an ECMA **script**, and an **array** element with arguments
Attributes	**name**: The name of the variable to change **expr**: (optional): The new value for the expression. **select**: (optional): An XPath expression for the new value. **globalto** (optional): "true" if the destination variable is in the global space **globalfrom** (optional, for XPath instructions): "true" if the XPath in the **select** attribute should be applied to the global space.
Applies to	None

Each dialogue in an application has its own variable space, in addition to the global variable space. There are three different ways of using a **set** instruction to change a variable value:

- The **expr** attribute. This can contain a raw value, such as a string or number, or a string containing variable names

```
<set var="numberCopyVar" expr="$anumber"/>
```

- The **select** attribute. This uses an XPath expression for the new value. This is an excellent way to work with DOM structures. There is more detail about using XPath with FXML in the next section.

```
<set name="xpathset1" select="name(result/interpretation/instance/test)"/>
```

- An ECMA script. The result of the script is stored in the variable given in the **name** attribute of the **set** element. Values can be passed into the script with **var** and **array** child elements, where each element associates a local name with a variable outside of the script. This is an excellent way to do complex string manipulation and calculations.

```
<set oper="js">
  <var name="testvar" expr="$conditionVar"/>
  <var name="testvar2" expr="$conditionVarReverse"/>
  <script>
      testvar == 123 && testvar2 == 321;
  </script>
</ucond>
```

5.2.2 Checking conditions: `conditions`, `cond`, and `ucond`

Annotation	conditions
Definition	A set of conditions. Can itself be used recursively as a condition.
Children	`cond`: (optional): A condition with two arguments. `ucond`: (optional): A condition with one argument. `conditions`: (optional): A set of conditions.
Attributes	`oper` (optional): This can be "and" or "or". Defaults to "and".
Applies to	None

Annotation	cond
Definition	A comparison condition
Children	None
Attributes	`oper` : This can be "eq", "gt", or "lt"
Applies to	None

Annotation	ucond
Definition	A unary condition
Children	• If `oper` is "js", arguments to an ECMA script and an ECMA script • If `oper` is "not", another condition (`cond`, `ucond`, or `conditions`)
Attributes	`oper` : This can be "not", "xpath", or "js" `global` (optional for XPath conditions): "true" if the XPath should be applied to the global variable space.
Applies to	None

These three elements are used throughout FXML for decision-making. They are used, for example, to determine if a context shift should occur, and to see if an RTN transition is traversable. As with the `set` instruction, ECMA script and XPath can both be used. For example, if we wished to see if a

counter has exceeded its maximum value, there are a couple of ways we could do that. The simplest is to use the greater-than function supported by FXML:

```
<conditions>
   <cond oper="gt" expr1="iterationCounter" expr2="$maxIteration"/>
</conditions>
```

alternately, we could pass the relevant values into an ECMA script for comparison.

```
<conditions>
   <ucond oper="js">
      <var name="localCounter" expr="$iterationCounter"/>
     <var name="localMax" expr="$maxIteration"/>
     <script>
      localCounter > localMax;
     </script>
   </ucond>
</conditions>
```

A `conditions` element can be a child of another `conditions` element, so that logical statements can be nested and combined.

5.2.3 Host Access Instructions: `http-access`, `jdbc-access`, and `file-access`

Annotation	**http-access** or **jdbc-access** or **file-access**
Definition	Calls an external data source or application
Children	None
Attributes	Common Parameters • `href` - A URL specifying either a database data source or a web site. • `resultset` - Reference to a local context variable that is used to return a string identifier (ID) for the root element of the result set. The resultset, which is a DOM tree, is stored in the local context. • `params` - A reference to a local context dictionary with instruction and data parameters. • `error` - A reference to a local context variable, which is used to store the error message text when there is a connection error. • `max` - Sets the maximum number of records that are returned. The default value for the 'max' parameter is "-1", which stands for all records. • `query` - A SQL query string. HTTP-specific request parameters: • `content_type` - Specifies the encoding of the HTTP POST parameters and also the content of a HTTP file access instruction. • `proxyhost`, `proxyport` - Proxy setting for HTTP connection through a proxy server. • `authentication` - Defines the authentication type. Can be either "BASIC" or "Other". • `login`, `password` - Authentication settings used only in combination with the 'authentication' parameter. • `connection_timeout` - Specifies the maximum time (in msec) for the instruction to establish a connection to a host. • `io_timeout` - Specifies the maximum time (in msec) for the instruction

	to wait for data. JDBC-specific request parameters: • `driver` - A jdbc database driver name. The 'driver' parameter must contain the exact path and class name of a JDBC driver. • `login` / `password` - User id and password for the database access. The default value for both parameters is an empty string (""), which is sent to the database when connecting.
Applies to	None

The database access instructions perform data access to a host using either a HTTP or a JDBC connection. Data returned from a host access is stored in the local context .

• `<http-access>`: Instruction to connect to Servlets, CGI Scripts or static WEB contents on a WEB server using HTTP.
• `<jdbc-access>`: Instruction to execute SQL query on a database using a JDBC connection.
• `<file-access>`: Instruction to read files.

The results of these instructions are stored as a DOM structure in a local variable, where they can be accessed by other instructions and conditions. In case of a connection error, the error message is stored as a text in the local context variable, which is referenced by the 'error' parameter.

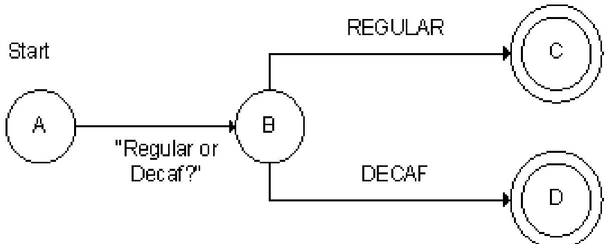
```
 <http-access
    href="http://wevs01.research.att.com/dbAccess/dbAccess"
    content_type="application/x-www-form-urlencoded"
    resultset="res"
    size="ressize"
    error="reserr" >

        select * from wevsbill.wevs_cdh

</http-access>
```

6. Flow Controller Syntax

The FXML syntax for flow control is open-ended. This document will describe two flow controller implementations for FXML, but this should not be taken to mean that development is limited to those two models. Within the IM framework, additional models may be developed to suit specific approaches to dialogue. The IM framework supports the nesting of flow controllers, so it is permitted for a single application to use multiple flow controllers types. This may find application where different parts of a dialogue call for different types of decision-making models.

6.1 The Recursive Transition Network (RTN) Flow Controller

The RTN controller is used to represent state machine control flow logic. Dialogues are defined in terms of states and transitions between those states. For example, if we had a routing application that directed users based on their response to a simple question, it could be described as an RTN with four nodes:



Where A is the start state, B is the input state, and C and D are both final states. In FXML, we might represent this RTN with the following code:

```
<fxml>
  <rtn name="coffee" start="A">
    <actiondefs>
      <actiondef name="typeQuery" text="Regular or Decaf?">
        <fragment name="VXML-grammar">
      <grammar src="CoffeeGrammar.std"/>
        </fragment>
      </actiondef>
    </actiondefs>

    <states>
      <state name="A">
        <transitions>
          <transition from="A" to="B" name="query">
            <actions>
              <action>typeQuery</action>
            </actions>
          </transition>
        </transitions>
       </state>
      <state name="B" pause="true">
        <transitions>
          <transition to="C" name="regular">
      <conditions>
        <cond oper="eq" expr1="result" expr2="REGULAR"/>
      </conditions>
          </transition>
          <transition to="D" name="decaf">
      <conditions>
        <cond oper="eq" expr1="result" expr2="DECAF"/>
      </conditions>
          </transition>
        </transitions>
       </state>
      <state name="C" final="true">
        <transitions>
        </transitions>
       </state>
      <state name="D" final="true">
        <transitions>
        </transitions>
       </state>
     </states>
   </rtn>
</fxml>
```

State "B" contains the attribute `pause`, which is set to "true". This is where the system is done with its turn, creates output for the user (in this case, the question "Regular or decaf?"), and allows the user to respond. In this example, the grammar used for this query has been simplified to single words ("REGULAR" and "DECAF") instead of a full XML document for the sake of clarity. Examples of conditions based on full XML input are given in Section 7. The variable that the conditions in this example check, "result", is where new user input is always stored.

6.1.1 RTN top-level element: `rtn`

Annotation	rtn
Definition	Defines one level of a recursive transition network
Children	`local` (optional): Local variables. `subdialogs` (optional): References to other dialogue definitions that are invoked in this file. `actiondefs` (optional): Definitions of actions used in this file. `states`: The states of the RTN. `transitions` (optional): The transitions of the RTN.
Attributes	• **All required**: ○ `name`: The ID of this dialogue. ○ `start`: The name of the start state. ○ `default`: The name of the default state. This state is visited if the current state does not have a traversable transition out of it.
Applies to	None

The `rtn` element is always an only child of the `fxml` element. The name of the RTN can be used by the IM and other dialogs to invoke it. This declaration can name two special states: the `start` state is self-explanatory. The `default` state is where the dialogue control goes if all of the transitions from the current state are untraversable. This can be used to catch unanticipated dialogue conditions. When the dialogue reaches a final state ("C" or "D" in the example, the dialogue immediately exits. If it was called by another dialogue, the other dialogue will have the opportunity to collect values from the RTN variables.

6.1.2 RTN state: `state`

Annotation	state
Definition	An RTN state
Children	`transitions` (optional): Transitions from this state. `enterstate` (optional): Instructions to be executed when the state is entered. `exitstate` (optional): Instructions to be executed when the state is exited. `entersubdialog` (optional): Instructions to be executed when the subdialog is entered. `exitsubdialog` (optional): Instructions to be executed when the subdialog returns.
Attributes	• `name`: The ID of this state. • `pause` (optional): "True" if the dialog should pause here for new input from the user. • `subdialog` (optional): The name of a subdialog to invoke in this state. • `final` (optional): "True" if the dialog returns when it reaches this state.
Applies to	None

This element is the equivalent of a state in a flow chart, as shown in the example for this section. It

is always a child of a `states` element. States are decision points where one of the outgoing transitions are chosen to continue the dialogue flow. Before making this decision, the state can execute instructions, pause for user input, or call a subdialogue. In the example for this section, state "B" (shown below) is a decision state that pauses for user input.

```
<state name="B" pause="true">
    <transitions>
      <transition to="C" name="regular">
  <conditions>
    <cond oper="eq" expr1="result" expr2="REGULAR"/>
  </conditions>
      </transition>
      <transition to="D" name="decaf">
  <conditions>
    <cond oper="eq" expr1="result" expr2="DECAF"/>
  </conditions>
      </transition>
    </transitions>
  </state>
```

To invoke a subdialog, assign the `subdialog` attribute the name of the flow controller definition that will be invoked. Before the subdialog takes control, the set of instructions in the `entersubdialog` child will be executed. These instructions can be used to copy values into the new dialogue. Likewise, when the subdialog is completed, the set of instructions in the `exitsubdialog` child will be executed,

6.1.3 RTN transition: `transition`

Annotation	transition
Definition	An RTN transition
Children	`actions` (optional): The actions that create output for the user. `conditions` (optional): Conditions that must be met before this transition is traversable. `instructions` (optional): Instructions that are executed when this transition is made.
Attributes	• `name`: The ID of this transition. • `from`: The name of the state where this transition originates • `to`: The name of the state where this transition ends. • `else` (optional): "True" if this transition is only used when all other transitions from the "from" state cannot be traversed. An "else" transition may not have conditions.
Applies to	None

This element describes a transition that links `state` elements. The two primary roles of this transition are to enforce the conditions in the call flow, and to add actions to the system output. The `actiondef` element (described in the previous section) provides the details of the action itself. The transition only needs to refer to the name of the action. Transitions are only traversable if the `conditions` child element can be evaluated as "true". If the transition has no `conditions` child element, and the `else` attribute is set to "true", then the transition is only traversable if no other transitions are.

From the RTN example given at the beginning of this section, this is a transition that adds an action ("typeQuery") to the system output:

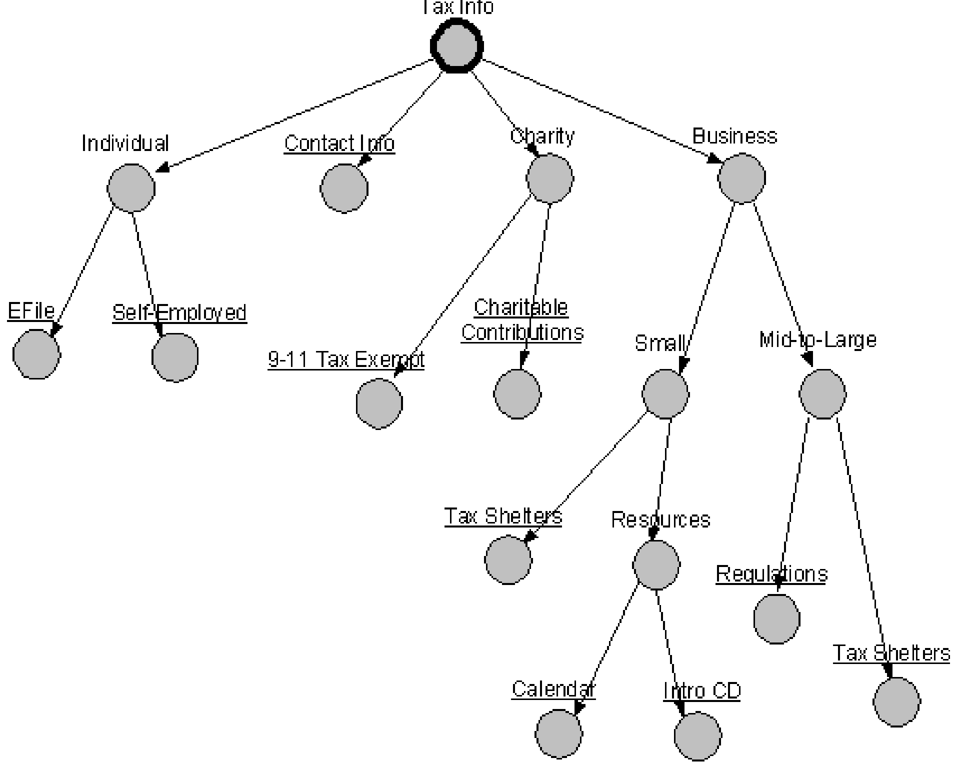
```
<transition from="A" to="B" name="query">
   <actions>
      <action>typeQuery</action>
   </actions>
</transition>
```

6.2 The Clarification Flow Controller

Dialogue systems are usually designed around a flowchart representation that guides the user through a series of subtasks, where each question permits a limited range of responses. This forces the user to use very limited language to be understood by the system. Although very common, this approach does not take advantage of information volunteered by the user which could speed up the process and improve the user experience.

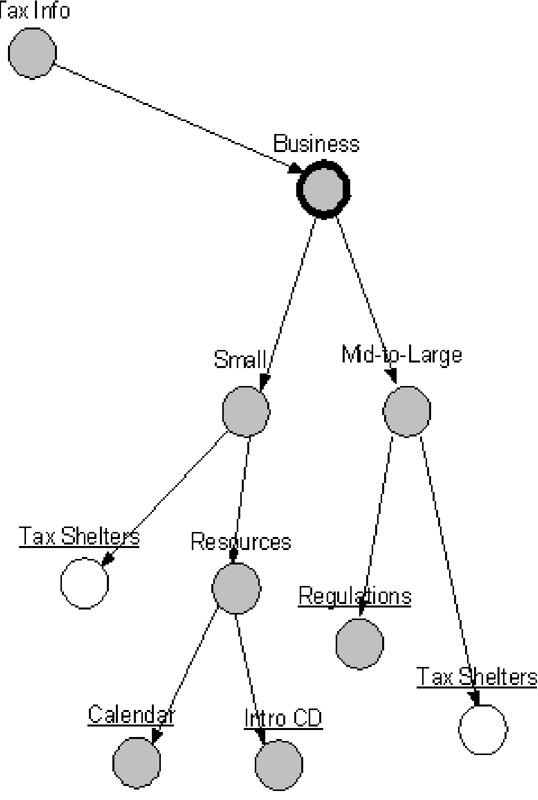
The Clarification FC operates on a compact representation of topic hierarchy . The algorithm it uses shortens the length of the dialogue by utilizing all the information offered by the user, whenever it is volunteered, and thereby allows the user to express themselves more naturally. The representation that this FC uses is a concept hierarchy that includes descriptions of the semantic categories the user can reference. The user input is matched to these descriptions, and then one or more system questions are retrieved from the tree to further discern the user needs. The system gradually navigates down the tree until it reaches a leaf. Each leaf indicates a specific topic that the system can address.

For example, consider an application for handling IRS information requests. The concept hierarchy might look like this:



Where the system always starts at the root of the tree ("Tax Info") and, based on user input at each

node, navigates down the tree to one of the leaves. If this was handled as an RTN, the system would traverse one transition at a time. The clarification algorithm tries to speed up this process by allowing user initiative to guide the system. For example, if the user starts out by saying that they are looking for a tax shelter, this narrows down the possible final nodes to the tax shelter leafs. This prunes off most of the tree like so:



Where the heavily outlined node ("Business") is the new focus of the dialogue. From this point, a single query from the system can discern between the two final possibilities.

In FXML, this application would be represented with a Clarification FC like this:

```
<fxml>
  <clarify name="IRSCall">
       <local/>
     <actiondefs>
        <actiondef name="TAX_INFO" text="Hello, what tax information would you like?"/>
        <actiondef name="INDIVIDUAL_INFO_PROMPT" text="I can give you EFiling information, or tell
        <actiondef name="EFILE_INFO_PROMPT" text="EFiling will get you your refund more quickly."/>
...
     </actiondefs>
     <nodes startfocus="TAX_INFO">
        <node name="TAX_INFO">
           <actions>
              <action>TAX_INFO</action>
           </actions>
        </node>
...
        <node name="BUSINESS_INFO" parent="TAX_INFO">
           <cond oper="eq" expr1="$result" expr2="Business"/>
           <actions>
              <!-- <action>BUSINESS_INFO_PROMPT</action>  -->
           </actions>
        </node>
        <node name="SMALL_BUSINESS_INFO" parent="BUSINESS_INFO" >
           <cond oper="eq" expr1="$result" expr2="SmallBusiness"/>
           <actions>
```

```
            <action>SMALL_BUSINESS_INFO_PROMPT</action>
         </actions>
      </node>
      <node name="MID_LARGE_BUSINESS_INFO" parent="BUSINESS_INFO" >
         <cond oper="eq" expr1="$result" expr2="MLBusiness"/>
         <actions>
            <action>MID_LARGE_BUSINESS_INFO_PROMPT</action>
         </actions>
      </node>
      <node name="SMALL_BIZ_SHELTER_INFO" parent="SMALL_BUSINESS_INFO" >
         <cond oper="eq" expr1="$result" expr2="Shelter"/>
         <actions>
            <action>SMALL_BIZ_SHELTER_INFO_PROMPT</action>
         </actions>
      </node>
      <node name="SMALL_BIZ_RESOURCES" parent="SMALL_BUSINESS_INFO" >
         <cond oper="eq" expr1="$result" expr2="SBizResources"/>
         <actions>
            <action>SMALL_BIZ_RESOURCES_PROMPT</action>
         </actions>
      </node>
      <node name="SMALL_BIZ_CALENDAR" parent="SMALL_BIZ_RESOURCES" >
         <cond oper="eq" expr1="$result" expr2="SBCalendar"/>
         <actions>
            <action>SMALL_BIZ_CALENDAR_PROMPT</action>
         </actions>
      </node>
      <node name="SMALL_BIZ_CD" parent="SMALL_BIZ_RESOURCES" >
         <cond oper="eq" expr1="$result" expr2="SBCD"/>
         <actions>
            <action>SMALL_BIZ_CD_PROMPT</action>
         </actions>
      </node>
      <node name="ML_REGULATION_INFO" parent="MID_LARGE_BUSINESS_INFO" >
         <cond oper="eq" expr1="$result" expr2="MLRegulation"/>
         <actions>
            <action>ML_REGULATION_INFO_PROMPT</action>
         </actions>
      </node>
      <node name="ML_SHELTER_INFO" parent="MID_LARGE_BUSINESS_INFO">
         <cond oper="eq" expr1="$result" expr2="Shelter"/>
         <actions>
            <action>ML_SHELTER_INFO_PROMPT</action>
         </actions>
      </node>
   </nodes>
  </clarify>
</fxml>
```

Some nodes and action definitions have been omitted for brevity. More examples with this application, and details of this algorithm, can be found in [1].

Sample exchange 1: System Initiative

System: This is the IRS help system. What can I do for you?
User: I need Tax Information.
System: I can help you with contact info, individual returns, business returns, or charitable organization returns.
User: Individual returns.
System: Would you like to hear about e-filing or self-employment?
User: Electronic filing.

Sample exchange 2: Disambiguation

System: This is the IRS help system. What can I do for you?
User: I need to find out more about tax shelters.
System: Are you a small business, or a mid-to-large sized business?
User: A small business.

6.2.1 Clarification top-level element: `clarify`

Annotation	clarify
Definition	Defines a Clarification FC.
Children	`local` (optional): Local variables. `subdialogs` (optional): References to other dialogue definitions that are invoked in this file. `actiondefs` (optional): Definitions of actions used in this file. `nodes`: The nodes of the concept hierarchy
Attributes	`name`: The ID of this dialogue.
Applies to	None

This is the top element of a Clarification FC, and is always the child of the `fxml` element.

6.2.2 Clarification tree node: `node`

Annotation	node
Definition	A node in the hierarchy tree of a clarification flow controller.
Children	`actions` (optional): Transitions from this state. `conditions` (optional): Transitions from this state. `enterstate` (optional): Instructions to be executed when the node is entered. `exitstate` (optional): Instructions to be executed when the node is exited. `entersubdialog` (optional): Instructions to be executed when the subdialog is entered. `exitsubdialog` (optional): Instructions to be executed when the subdialog returns.
Attributes	`name`: The ID of this node. `parent`: The ID of the parent node of this node. `subdialog` (optional): The ID of a dialogue called from this node.
Applies to	None

The `node` element is similar to the `state` element in the RTN FC. The instruction sets and the `subdialog` attribute are used in exactly the same way. The `actions` and `conditions` children of the `node` element are more similar to the `transition` element in the RTN FC, but they operate differently.

When the Clarification FC stops for user input, there will be one node that is the current focus. This is the node that determines how the system should prompt the user. In some cases, the focus may not move between turns so, to avoid redundant prompts, more than one prompt may be defined for a node. The `actions` element describes what actions should be used as prompts from this node.

When new input is received, the Clarification FC checks all not yet relevant nodes in the current tree to see if they are now relevant, as defined by the `conditions` element. Once a node has been determined to be relevant it stays that way for the rest of the dialogue's life. As show in the example above, when multiple nodes are considered relevant it can force the FC to ask a question

that discerns between them.

7. XPath in FXML

FXML fully supports XPath 1.0 for working with structured XML input. New input is accessible as a global variable named "result", and XPath instructions and conditions can be used to access it to control dialogue flow. For example, take this simplified NLSML input for an ambiguously classified user utterance:

```
<result>
  <interpretation>
    <input>I'm trying to find out what you have as far as our account goes how does it stand</input
    <instance>I’m trying to find out what you have as far as our account goes how does it stand</ir
    <classes>
      <class name="Tell(Account_Status)" score="0.96"/>
      <class name="Tell(Account_Features)" score="0.90"/>
    </classes>
  </interpretation>
</result>
```

The NLSML is an interpretation of a speech utterance that includes possible categories that the utterance may fall into, with confidence scores for each. In this example, there are two confusable utterance types, Tell(Account_Status) and Tell(Account_Features), each with a high confidence score (>0.7) and with these scores very close to each other (<0.2). If we wish to identify this situation in an FXML application, it is easily described by an XPath condition:

```
<ucond oper="xpath"
   expr=”(count(//class[@score > $threshold ]) = 2) and
       ((//class [position() = 1]/@score - //class [position() = 2] /@score < $closeScore))”/>.
```

The "$" expressions refer to local or global context variables. This condition can be used to guide the dialogue to a disambiguation question like "do you want your account status or the account features?"

In the course of a dialogue, the Florence IM keeps a history of past inputs in a global variable. This permits the author to make similar conditions based on patterns of input over the course of the entire dialogue, such as the number of times the user has not responded or the number of times that a particular mode is used. The history is a DOM structure that can be accessed by XPATH expressions. Below is a simplified snapshot of a dialogue history with three turns stored within the tag `<history>` (the global variable used by the IM).

```
<history>
   <turns>
      <turn type="concrete">
         <result>
            <interpretation>
               <input>could you tell me the repair department phone number</input>
               <instance>could you tell me the <dept>REPAIR</dept> phone number</instance>
               <classes>
                  <class name="Req(Number)" score="0.94"/>
               </classes>
            </interpretation>
         </result>
      </turn>
      <turn type="discourse">
         <result> ...
               <instance>hello</instance> ...
      </turn>
      <turn type="command">
```

```
          <result>
             <interpretation>
                <input>could you repeat please</input>
                <instance>could you repeat that please</instance>
                <classes>
                   <class name="Repeat" score="0.98"/>
                </classes>
             </interpretation>
          </result>
       </turn>
    </turns>
 </history>
```

Within the **<turn>** element, the attribute 'type' specifies the category of the call-type, in terms of user's intentions. Typically call-types can be categorized as concrete when they convey information that contribute to the user request, as opposed to discourse or command types where the contribution is irrelevant to focus of the dialogue.

In the last turn in this example, the user asks the system to repeat the repairs department phone number announced in the first turn, but the information to repeat is not explicitly stated (this is called an anaphora reference). The IM must go to the history of the dialogue to find the turn that the user is referring to. This translates to the following XPath expression:

<set name="previousReference" select="//turn[last() and @type = 'concrete']//class [@score>0.7]/@name"/>

In this example, this instruction selects the first concrete call-type correctly and assigns the value "Req(Number)" to the "previousReference" context variable.

For a more detailed look at the use of XPath for discourse analysis, see [3].

8. FXML application examples

This section develops a multimodal "Hello, World" application in three stages, followed by demonstrations of how FXML accomplishes some simple dialogue tasks. In the first example, the user interacts directly with the Florence IM from a command line. In the second stage, we will give the application an HTML interface. This demonstrates how presentation details are kept independent from the control flow. The third part puts the Florence IM into a multimodal context, and give an example of an application that interacts through both voice and HTML.

8.1 Command line "Hello, World!"

Every Florence application requires an FXML configuration file. In our first example, this configuration file will be as simple as it gets:

```
 <fxml>
   <configuration>
     <dialogfile>HW-CL-toplevel.fxml</dialogfile>
     <nlu/>
     <output/>
   </configuration>
 </fxml>
```

The NLU element in FXML is used to direct the application to a pre-processor for the input. This configuration does not contain any pre-processor information, so input will go directly to the Florence IM. The output element in FXML is used to indicate an output template. No template information is given, so the action text will go directly to the console. The **dialogfile** element points to the top-level dialogue definition, **HW-CL-toplevel.fxml**:

```
<fxml>
  <rtn name="HW-toplevel" start="start">
    <actiondefs>
      <actiondef name="helloWorld" text="Hello, World!"/>
    </actiondefs>

    <states>      <state name="start">
        <transitions>
          <transition from="start" to="end" name="t1">
            <actions>
              <action>helloWorld</action>
            </actions>
          </transition>
        </transitions>
       </state>

      <state name="end" final="true"/>
     </states>
   </rtn>
</fxml>
```

This is a simple RTN consisting of two states and a transition between them. The "start" attribute of the RTN element indicates the start state. The "Hello, World" action is defined outside of the states, and is invoked in the transition. This dialogue will say hello, and then end:

```
C:\docs\w3c\HW-CL>java florence.FlorenceTester HW-CL-config.fxml .
Florence version 1.18, build 3
Florence says:
Hello, World!
```

The essential control flow elements in this example (the transitions and states), will not change with the addition of other modalities.

8.2 HTML "Hello, World!"

In the second "Hello, World!" example we will interact with the user through an HTML page. This requires a change to the `output` element in the configuration file:

```
<fxml>
    <configuration>
        <dialogfile>HW-CL-toplevel.fxml</dialogfile>
        <nlu/>
        <output template="HW.html"/>
    </configuration>
</fxml>
```

Now, instead of returning the text value of the action invoked by the control flow, the Florence IM will return a template filled in with values from the action. In this case, the application is accessed through an HTML page that asks a servlet running the Florence IM for new HTML. The template for this output looks like this:

```
<html>
    <body>
        $TEXT
    </body>
</html>
```

For demonstration purposes this template is very simple, but it can be fleshed out to create a more elaborate web page. This is just a template that will be filled in by the Florence IM and submitted to a web browser, so it is only limited by the HTML syntax. This template uses a special variable

$TEXT, which will be replaced with the value of the text attribute of the action (or actions) that are invoked. In this case the page will show the text, "Hello World!".

Except for a small change to the configuration file, the changes necessary to migrate our application from console output to HTML occur outside of FXML. Similarly, very few FXML changes are needed to make this application voice-enabled.

8.3 HTML + VXML "Hello, World!"

The preponderance of work needed to make an FXML application multimodal occurs outside of the FXML. All of the tasks which synchronize modalities and solicit input have to occur outside of the Florence IM. To assist with this synchronization, the Florence IM accepts a session ID along with user input, and this ID can be used in the template.

Multimodal output is accomplished by creating output files for each modality (in this case, HTML and VXML) and leaving the synchronization task to the primary modality (in this case, HTML). The output template contains templates for both of these modalities. The servlet will split the completed template and save each part to an appropriate file type. For our example, the mixed template looks like this:

```
<html>
    <body ONLOAD="loadVXML('$JSESSIONID')">
        $TEXT
    </body>
</html>

<vxml>
    <form>
        <block>
            <prompt>
                $TEXT
            </prompt>
        </block>
    </form>
</vxml>
```

Needless to say, some details are omitted here: the media server settings, the HTML scripts, etc, that perform the multimodal synchronization. These details are kept encapsulated in the templates, away from the FXML control flow.

8.4 "Hello, World!" with tricks.

These are a few short examples of elementary dialogue tasks in FXML.

8.4.1 Hello loop.

First, a simple loop. In this example (augmented from 8.1, the command line "Hello, World!"), the control flow will pause after saying hello, and then repeat itself after receiving input from the user. The config file (not shown) remains identical to the one in 8.1.

```
<fxml>
  <rtn name="HW-toplevel" start="start">
```

```
    <actiondefs>
      <actiondef name="helloWorld" text="Hello, World!"/>
    </actiondefs>

    <states>
   <state name="start">
        <transitions>
          <transition from="start" to="repeat" name="t1">
            <actions>
              <action>helloWorld</action>
            </actions>
          </transition>
        </transitions>
       </state>
       <state name="repeat" pause="true">
        <transitions>
          <transition from="repeat" to="start" name="t2"/>
        </transitions>
       </state>
    </states>
  </rtn>
</fxml>
```

This example includes the "pause" attribute in the second state. This instructs the IM to enact any actions is has queued, and to wait for new input from the user. In this case, the input is ignored and the cycle repeats.

8.4.2 NLSML Parsing.

This example will loop as long as the user is silent. When the user replies to the IM, the dialogue will end.

```
<fxml>
  <rtn name="HW-toplevel" start="start">
    <actiondefs>
      <actiondef name="helloWorld" text="Hello, World!"/>
    </actiondefs>

    <states>
   <state name="start">
        <transitions>
          <transition from="start" to="listen" name="t1">
            <actions>
              <action>helloWorld</action>
            </actions>
          </transition>
        </transitions>
       </state>
       <state name="listen" pause="true">
        <transitions>
          <transition from="listen" to="start" name="t2" else="true"/>
          <transition from="listen" to="end" name="t3">
      <conditions>
         <ucond oper="xpath" expr="result/interpretation/input/noinput"/>
      </conditions>
      </transition>
        </transitions>
       </state>
       <state name="end" final="true"/>
    </states>
  </rtn>
</fxml>
```

The XPath condition that detects silence uses the "result" value, which is available to all instructions and conditions. This example also uses the "else" attribute of a transition. There may be one "else" transition per transition set, and that transition is selected if none of the others are traversable.

8.4.3 HTML input.

This example builds on 8.2. In addition to using HTML to say "Hello, World!", it also presents a button which tells the Florence IM to say good-bye. This is the output template:

```
<html>
    <body>
        <form method="post" action="/FlorenceServlet" >
        $TEXT
        <input type=submit name="inputButton" value="BYE">
    </body>
</html>
```

In this example (and in 8.2), the Florence IM is run as a servlet. The HTML generated by the IM is displayed in a browser by requesting a page from the servlet, like this:

```
http://FlorenceServer.research.att.com:8080/FlorenceServlet
```

As you can see in the output template, the page generated by the IM will get a new page from the servlet by posting when there is new input.

In this case, instead of using NLSML (although this is still an option) the servlet will send the value of the button to the Florence IM as text. If the text is "BYE", the HTML page will show a new message. Otherwise (for example, if we had other buttons or a timer in the HTML page to detect no input), the initial message will be repeated. This logic also handles the initial call, where there is no input.

```
<fxml>
  <rtn name="HW-toplevel" start="start">
    <actiondefs>
      <actiondef name="helloWorld" text="Hello, World!"/>
      <actiondef name="goodbyeCruelWorld" text="Goodbye, World!"/>
    </actiondefs>

    <states>
   <state name="start">
        <transitions>
          <transition from="start" to="listen" name="t1">
            <actions>
              <action>helloWorld</action>
            </actions>
          </transition>
        </transitions>
       </state>
       <state name="listen" pause="true">
        <transitions>
          <transition from="listen" to="start" name="t2" else="true"/>
          <transition from="listen" to="end" name="t3">
        <conditions>
         <cond oper="eq" expr1="result" expr1="BYE"/>
        </conditions>
            <actions>
              <action>goodbyeCruelWorld</action>
            </actions>
      </transition>
        </transitions>
       </state>
       <state name="end" final="true"/>
    </states>
  </rtn>
</fxml>
```

Appendices

A. Florence XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2006 sp2 U (http://www.altova.com) by Charles Lewis (W3C) -->
<!--W3C Schema generated by XMLSpy v2006 sp2 U (http://www.altova.com)-->
<xs:schema targetNamespace="http://research.att.com/e-contact/florence" xmlns:xs="http:/
   <xs:element name="fxml">
      <xs:annotation>
         <xs:documentation>Top level tag</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:choice>
            <xs:element ref="configuration"/>
            <xs:element ref="rtn"/>
            <xs:element ref="clarify"/>
            <xs:element ref="global"/>
         </xs:choice>
         <xs:attribute name="version" use="required" fixed="2.0">
            <xs:simpleType>
               <xs:restriction base="xs:decimal">
                  <xs:minInclusive value="0.01"/>
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
      </xs:complexType>
   </xs:element>
   <xs:element name="global">
      <xs:annotation>
         <xs:documentation>The global variables</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="var"/>
            <xs:element ref="array"/>
            <xs:element ref="dictionary"/>
         </xs:choice>
      </xs:complexType>
      <xs:key name="globalVADK">
         <xs:selector xpath="var | array | dictionary"/>
         <xs:field xpath="@name"/>
      </xs:key>
   </xs:element>
   <xs:element name="configuration">
      <xs:annotation>
         <xs:documentation>Describes the run-time settings for this FXML application</xs
      </xs:annotation>
      <xs:complexType>
         <xs:all>
            <xs:element ref="dialogfile"/>
            <xs:element ref="nlu" minOccurs="0"/>
            <xs:element ref="output"/>
            <xs:element ref="logfile" minOccurs="0"/>
         </xs:all>
         <xs:attribute name="logfile" type="xs:anyURI"/>
         <xs:attribute name="loglevel" type="logging" default="FATAL"/>
         <xs:attribute name="debuglevel" type="logging" default="FATAL"/>
         <xs:attribute name="consolemessages" type="xs:boolean" default="false"/>
         <xs:attribute name="checkvars" type="xs:boolean" default="false"/>
         <xs:attribute name="globals" type="xs:anyURI"/>
         <xs:attribute name="afterdialog" type="xs:anyURI"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="dialogfile">
      <xs:annotation>
         <xs:documentation>A reference to another FXML file that is referred to in this
      </xs:annotation>
      <xs:simpleType>
```

```
      <xs:restriction base="xs:anyURI">
        <xs:minLength value="1"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:element name="nlu">
    <xs:annotation>
      <xs:documentation>Connection info for the input pre-processor</xs:documentation
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="threshold" type="xs:decimal"/>
      <xs:attribute name="host" type="xs:anyURI"/>
      <xs:attribute name="port" type="xs:positiveInteger"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="output">
    <xs:annotation>
      <xs:documentation>Information for generating the output</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="template" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="logfile">
    <xs:annotation>
      <xs:documentation>URI for a logfile</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:anyURI">
          <xs:attribute name="level" type="logging"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="clarify">
    <xs:annotation>
      <xs:documentation>A Clarification FC</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:all>
        <xs:element ref="local" minOccurs="0"/>
        <xs:element ref="subdialogs" minOccurs="0"/>
        <xs:element ref="actiondefs" minOccurs="0"/>
        <xs:element ref="nodes"/>
      </xs:all>
      <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
    </xs:complexType>
    <xs:key name="clNK">
      <xs:selector xpath="./nodes/node"/>
      <xs:field xpath="@name"/>
    </xs:key>
    <xs:keyref name="clNsRef" refer="clNK">
      <xs:selector xpath="./nodes"/>
      <xs:field xpath="@startfocus"/>
    </xs:keyref>
    <xs:keyref name="clNRef" refer="clNK">
      <xs:selector xpath="./nodes/node"/>
      <xs:field xpath="@parent"/>
    </xs:keyref>
  </xs:element>
  <xs:element name="nodes">
    <xs:annotation>
      <xs:documentation>A collection of Node elements</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="node"/>
      </xs:sequence>
      <xs:attribute name="startfocus" type="xs:string"/>
    </xs:complexType>
  </xs:element>
```

```
<xs:element name="node">
   <xs:annotation>
      <xs:documentation>A node in the Clarification FC taxonomy</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:all>
         <xs:element ref="actions" minOccurs="0"/>
         <xs:element ref="conditions" minOccurs="0"/>
         <xs:element ref="enterstate" minOccurs="0"/>
         <xs:element ref="entersubdialog" minOccurs="0"/>
         <xs:element ref="exitstate" minOccurs="0"/>
         <xs:element ref="exitsubdialog" minOccurs="0"/>
      </xs:all>
      <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
      <xs:attribute name="parent" type="xs:NMTOKEN"/>
      <xs:attribute name="pause" type="xs:boolean" default="true"/>
      <xs:attribute name="subdialog" type="xs:string"/>
   </xs:complexType>
</xs:element>
<xs:element name="rtn">
   <xs:annotation>
      <xs:documentation>An RTN FC</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:all>
         <xs:element ref="local" minOccurs="0"/>
         <xs:element ref="subdialogs" minOccurs="0"/>
         <xs:element ref="actiondefs" minOccurs="0"/>
         <xs:element ref="states"/>
         <xs:element ref="transitions" minOccurs="0"/>
      </xs:all>
      <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
      <xs:attribute name="start" type="xs:NMTOKEN" use="required"/>
      <xs:attribute name="default" type="xs:NMTOKEN" use="required"/>
   </xs:complexType>
   <xs:key name="rtnSK">
      <xs:selector xpath="./states/state"/>
      <xs:field xpath="@name"/>
   </xs:key>
   <xs:keyref name="rtnSSRef" refer="rtnSK">
      <xs:selector xpath="."/>
      <xs:field xpath="@start"/>
   </xs:keyref>
   <xs:keyref name="rtnDSRef" refer="rtnSK">
      <xs:selector xpath="."/>
      <xs:field xpath="@default"/>
   </xs:keyref>
   <xs:keyref name="rtnCSRef" refer="rtnSK">
      <xs:selector xpath="./contextshifts/contextshift"/>
      <xs:field xpath="@to"/>
   </xs:keyref>
   <xs:key name="rtnVADK">
      <xs:selector xpath="./local/var | ./local/dictionary | ./local/array"/>
      <xs:field xpath="@name"/>
   </xs:key>
</xs:element>
<xs:element name="actiondefs">
   <xs:annotation>
      <xs:documentation>A collection of action definitions</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="actiondef"/>
      </xs:sequence>
   </xs:complexType>
   <xs:key name="rtnAK">
      <xs:selector xpath="./actiondef"/>
      <xs:field xpath="@name"/>
   </xs:key>
</xs:element>
<xs:element name="actiondef">
   <xs:annotation>
```

```
        <xs:documentation>An action definition</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequencemaxOccurs="unbounded">
            <xs:element ref="fragment"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
        <xs:attribute name="text" type="xs:string"/>
        <xs:attribute name="args" type="xs:string"/>
    </xs:complexType>
</xs:element>
<xs:element name="fragment">
    <xs:complexType>
        <xs:sequence>
            <xs:any processContents="skip" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="name" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="actions">
    <xs:annotation>
        <xs:documentation>A collection of actions</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence maxOccurs="unbounded">
            <xs:element ref="action"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="action">
    <xs:annotation>
        <xs:documentation>An action taken in the call flow</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute name="args" type="xs:string"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:element>
<xs:element name="console" type="xs:string">
    <xs:annotation>
        <xs:documentation>A debugging command that writes to the console window</xs:doc
    </xs:annotation>
</xs:element>
<xs:element name="conditions">
    <xs:annotation>
        <xs:documentation>A collection of conditions</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="conditions"/>
            <xs:element ref="cond"/>
            <xs:element ref="ucond"/>
        </xs:choice>
        <xs:attribute name="oper" type="logicalBinaryOperator" use="optional" default="
    </xs:complexType>
</xs:element>
<xs:element name="cond">
    <xs:annotation>
        <xs:documentation>A condition</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:attribute name="oper" type="comparisonBinaryOperator" use="required"/>
        <xs:attribute name="expr1" type="xs:string" use="required"/>
        <xs:attribute name="expr2" type="xs:string" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="ucond">
    <xs:annotation>
        <xs:documentation>A condition with only one argument</xs:documentation>
```

```
      </xs:annotation>
      <xs:complexType>
         <xs:attribute name="oper" type="unaryOperator" use="required"/>
         <xs:attribute name="expr" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="contextshift">
      <xs:annotation>
         <xs:documentation>A context shift</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:attribute name="xpath" type="xs:string" use="required"/>
         <xs:attribute name="to" type="xs:NMTOKEN" use="required"/>
         <xs:attribute name="confirm" type="xs:boolean" default="false"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="enterstate">
      <xs:annotation>
         <xs:documentation>A collection of instructions to be executed when an RTN state
      </xs:annotation>
      <xs:complexType>
         <xs:choice maxOccurs="unbounded">
            <xs:element ref="set" minOccurs="0"/>
            <xs:element ref="increment" minOccurs="0"/>
            <xs:element ref="decrement" minOccurs="0"/>
            <xs:element ref="log" minOccurs="0"/>
            <xs:element ref="alarm" minOccurs="0"/>
            <xs:element ref="sqlquery" minOccurs="0"/>
            <xs:element ref="http-access" minOccurs="0"/>
            <xs:element ref="getdata" minOccurs="0"/>
            <xs:element ref="console" minOccurs="0"/>
         </xs:choice>
      </xs:complexType>
   </xs:element>
   <xs:element name="exitstate">
      <xs:annotation>
         <xs:documentation>A collection of instructions to be executed when an RTN state
      </xs:annotation>
      <xs:complexType>
         <xs:choice maxOccurs="unbounded">
            <xs:element ref="set" minOccurs="0"/>
            <xs:element ref="increment" minOccurs="0"/>
            <xs:element ref="decrement" minOccurs="0"/>
            <xs:element ref="log" minOccurs="0"/>
            <xs:element ref="alarm" minOccurs="0"/>
            <xs:element ref="sqlquery" minOccurs="0"/>
            <xs:element ref="http-access" minOccurs="0"/>
            <xs:element ref="getdata" minOccurs="0"/>
            <xs:element ref="console" minOccurs="0"/>
         </xs:choice>
      </xs:complexType>
   </xs:element>
   <xs:element name="entersubdialog">
      <xs:annotation>
         <xs:documentation>A collection of instructions to be executed when a subdialogu
      </xs:annotation>
      <xs:complexType>
         <xs:choice maxOccurs="unbounded">
            <xs:element ref="copy"/>
            <xs:element ref="set" minOccurs="0"/>
            <xs:element ref="increment" minOccurs="0"/>
            <xs:element ref="decrement" minOccurs="0"/>
            <xs:element ref="log" minOccurs="0"/>
            <xs:element ref="alarm" minOccurs="0"/>
            <xs:element ref="sqlquery" minOccurs="0"/>
            <xs:element ref="http-access" minOccurs="0"/>
            <xs:element ref="getdata" minOccurs="0"/>
            <xs:element ref="console" minOccurs="0"/>
         </xs:choice>
      </xs:complexType>
   </xs:element>
   <xs:element name="exitsubdialog">
```

```
      <xs:annotation>
         <xs:documentation>A collection of instructions to be executed when a subdialogu
      </xs:annotation>
      <xs:complexType>
         <xs:choice maxOccurs="unbounded">
            <xs:element ref="copy"/>
            <xs:element ref="set" minOccurs="0"/>
            <xs:element ref="increment" minOccurs="0"/>
            <xs:element ref="decrement" minOccurs="0"/>
            <xs:element ref="log" minOccurs="0"/>
            <xs:element ref="alarm" minOccurs="0"/>
            <xs:element ref="sqlquery" minOccurs="0"/>
            <xs:element ref="http-access" minOccurs="0"/>
            <xs:element ref="getdata" minOccurs="0"/>
            <xs:element ref="console" minOccurs="0"/>
         </xs:choice>
      </xs:complexType>
   </xs:element>
   <xs:element name="instructions">
      <xs:annotation>
         <xs:documentation>A collection of instructions in a call flow</xs:documentation
      </xs:annotation>
      <xs:complexType>
         <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="set"/>
            <xs:element ref="increment"/>
            <xs:element ref="decrement"/>
            <xs:element ref="log"/>
            <xs:element ref="alarm"/>
            <xs:element ref="sqlquery"/>
            <xs:element ref="http-access"/>
            <xs:element ref="getdata"/>
            <xs:element ref="console" minOccurs="0"/>
         </xs:choice>
      </xs:complexType>
   </xs:element>
   <xs:element name="set">
      <xs:annotation>
         <xs:documentation>An instruction to change a variable value</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:attribute name="name" type="xs:string" use="required"/>
         <xs:attribute name="expr" type="xs:string" use="optional"/>
         <xs:attribute name="select" type="xs:string" use="optional"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="copy">
      <xs:annotation>
         <xs:documentation>An instruction to copy a variable between dialogues</xs:docum
      </xs:annotation>
      <xs:complexType>
         <xs:attribute name="action" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="increment">
      <xs:annotation>
         <xs:documentation>An instruction to increment a variable</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:attribute name="var" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="decrement">
      <xs:annotation>
         <xs:documentation>An instruction to decrement a variable</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:attribute name="var" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="log">
      <xs:annotation>
```

```
            <xs:documentation>An instruction to write a log entry</xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:simpleContent>
               <xs:extension base="xs:string">
                  <xs:attribute name="msg" type="logging" default="INFO"/>
                  <xs:attribute name="category" type="xs:string" default="LOG_INSTRUCTION"/
               </xs:extension>
            </xs:simpleContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="alarm">
         <xs:annotation>
            <xs:documentation>An instruction that creates an alarm from the application</xs
         </xs:annotation>
         <xs:complexType>
            <xs:attribute name="category" type="xs:string" default="LOG_INSTRUCTION"/>
         </xs:complexType>
      </xs:element>
      <xs:element name="sqlquery">
         <xs:annotation>
            <xs:documentation>An instruction for accessing a data source</xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:simpleContent>
               <xs:extension base="xs:string">
                  <xs:attribute name="href" type="xs:anyURI" use="required"/>
                  <xs:attribute name="proxyhost" type="xs:anyURI" use="optional"/>
                  <xs:attribute name="proxyport" type="xs:integer" use="optional"/>
                  <xs:attribute name="login" type="xs:string" use="optional"/>
                  <xs:attribute name="password" type="xs:string" use="optional"/>
                  <xs:attribute name="driver" type="xs:string" use="optional"/>
                  <xs:attribute name="max" type="xs:string" use="optional"/>
                  <xs:attribute name="resultset" type="xs:string" use="optional"/>
                  <xs:attribute name="size" type="xs:string" use="optional"/>
                  <xs:attribute name="error" type="xs:string" use="optional"/>
                  <xs:attribute name="authentication" type="authentication" use="optional"/
                  <xs:attribute name="params" type="xs:string" use="optional"/>
                  <xs:attribute name="io_timeout" type="xs:nonNegativeInteger" use="optiona
                  <xs:attribute name="connection_timeout" type="xs:nonNegativeInteger" use=
                  <xs:attribute name="content_type" type="xs:string" use="optional" default
               </xs:extension>
            </xs:simpleContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="http-access">
         <xs:annotation>
            <xs:documentation>An instruction for accessing external components through an H
         </xs:annotation>
         <xs:complexType>
            <xs:simpleContent>
               <xs:extension base="xs:string">
                  <xs:attribute name="href" type="xs:anyURI" use="required"/>
                  <xs:attribute name="proxyhost" type="xs:anyURI" use="optional"/>
                  <xs:attribute name="proxyport" type="xs:integer" use="optional"/>
                  <xs:attribute name="login" type="xs:string" use="optional"/>
                  <xs:attribute name="password" type="xs:string" use="optional"/>
                  <xs:attribute name="max" type="xs:string" use="optional"/>
                  <xs:attribute name="resultset" type="xs:string" use="optional"/>
                  <xs:attribute name="size" type="xs:string" use="optional"/>
                  <xs:attribute name="error" type="xs:string" use="optional"/>
                  <xs:attribute name="authentication" type="authentication" use="optional"/
                  <xs:attribute name="params" type="xs:string" use="optional"/>
                  <xs:attribute name="io_timeout" type="xs:nonNegativeInteger" use="optiona
                  <xs:attribute name="connection_timeout" type="xs:nonNegativeInteger" use=
                  <xs:attribute name="content_type" type="xs:string" use="optional" default
               </xs:extension>
            </xs:simpleContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="getdata">
         <xs:complexType>
```

```
      <xs:attribute name="rs" type="xs:string" use="required"/>
      <xs:attribute name="result" type="xs:string" use="required"/>
      <xs:attribute name="error" type="xs:string"/>
      <xs:attribute name="op" type="getDataAction" use="required"/>
      <xs:attribute name="param" type="xs:string"/>
   </xs:complexType>
</xs:element>
<xs:element name="local">
   <xs:annotation>
      <xs:documentation>A set of variable definitions</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="var"/>
         <xs:element ref="array"/>
         <xs:element ref="dictionary"/>
      </xs:choice>
   </xs:complexType>
</xs:element>
<xs:element name="array">
   <xs:annotation>
      <xs:documentation>An array of values</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="value"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:string"/>
   </xs:complexType>
</xs:element>
<xs:element name="dictionary">
   <xs:annotation>
      <xs:documentation>A dictionary of values</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="var"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:string"/>
   </xs:complexType>
</xs:element>
<xs:element name="var">
   <xs:annotation>
      <xs:documentation>A variable definition</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="expr" type="xs:string"/>
   </xs:complexType>
</xs:element>
<xs:element name="value">
   <xs:annotation>
      <xs:documentation>The value of a variable</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:attribute name="expr" type="xs:string"/>
   </xs:complexType>
</xs:element>
<xs:element name="states">
   <xs:annotation>
      <xs:documentation>A collection of states</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
         <xs:element ref="state"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="state">
   <xs:annotation>
      <xs:documentation>An RTN state</xs:documentation>
   </xs:annotation>
```

```
<xs:complexType>
   <xs:all>
      <xs:element ref="enterstate" minOccurs="0"/>
      <xs:element ref="entersubdialog" minOccurs="0"/>
      <xs:element ref="exitstate" minOccurs="0"/>
      <xs:element ref="exitsubdialog" minOccurs="0"/>
      <xs:element ref="transitions" minOccurs="0"/>
   </xs:all>
   <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
   <xs:attribute name="pause" type="xs:boolean" default="true"/>
   <xs:attribute name="subdialog" type="xs:string"/>
   <xs:attribute name="final" type="xs:boolean" default="false"/>
</xs:complexType>
</xs:element>
<xs:element name="transitions">
   <xs:annotation>
      <xs:documentation>A collection of transitions</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
         <xs:element ref="transition"/>
      </xs:sequence>
   </xs:complexType>
   <xs:key name="rtnTK">
      <xs:selector xpath="./transition"/>
      <xs:field xpath="@name"/>
   </xs:key>
   <xs:keyref name="rtnTFSRef" refer="rtnSK">
      <xs:selector xpath="./transition"/>
      <xs:field xpath="@from"/>
   </xs:keyref>
   <xs:keyref name="rtnTTSRef" refer="rtnSK">
      <xs:selector xpath="./transition"/>
      <xs:field xpath="@to"/>
   </xs:keyref>
</xs:element>
<xs:element name="transition">
   <xs:annotation>
      <xs:documentation>An RTN transition</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:all>
         <xs:element ref="actions" minOccurs="0"/>
         <xs:element ref="conditions" minOccurs="0"/>
         <xs:element ref="instructions" minOccurs="0"/>
      </xs:all>
      <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
      <xs:attribute name="from" type="xs:NMTOKEN" use="required"/>
      <xs:attribute name="to" type="xs:NMTOKEN" use="required"/>
      <xs:attribute name="else" type="xs:boolean" default="false"/>
   </xs:complexType>
</xs:element>
<xs:element name="subdialogs">
   <xs:annotation>
      <xs:documentation>References to the subdialogs that will be used in an FXML dia
   </xs:annotation>
   <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="dialogfile"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:simpleType name="logicalBinaryOperator">
   <xs:restriction base="xs:string">
      <xs:enumeration value="and"/>
      <xs:enumeration value="nand"/>
      <xs:enumeration value="nor"/>
      <xs:enumeration value="or"/>
      <xs:enumeration value="xor"/>
   </xs:restriction>
</xs:simpleType>
<xs:simpleType name="comparisonBinaryOperator">
```

```
            <xs:restriction base="xs:string">
               <xs:enumeration value="eq"/>
               <xs:enumeration value="ge"/>
               <xs:enumeration value="gt"/>
               <xs:enumeration value="le"/>
               <xs:enumeration value="lt"/>
               <xs:enumeration value="ne"/>
            </xs:restriction>
         </xs:simpleType>
         <xs:simpleType name="unaryOperator">
            <xs:restriction base="xs:string">
               <xs:enumeration value="not"/>
               <xs:enumeration value="xpath"/>
               <xs:enumeration value="slu"/>
            </xs:restriction>
         </xs:simpleType>
         <xs:simpleType name="logging">
            <xs:restriction base="xs:string">
               <xs:enumeration value="INFO"/>
               <xs:enumeration value="DEBUG"/>
               <xs:enumeration value="WARN"/>
               <xs:enumeration value="ERROR"/>
               <xs:enumeration value="FATAL"/>
            </xs:restriction>
         </xs:simpleType>
         <xs:simpleType name="getDataAction">
            <xs:restriction base="xs:string">
               <xs:enumeration value="GETELEMENT"/>
               <xs:enumeration value="GETNUMELEMENTS"/>
               <xs:enumeration value="GETATTRIBUTE"/>
               <xs:enumeration value="GETNUMATTTRIBUTES"/>
               <xs:enumeration value="GETCONTENT"/>
            </xs:restriction>
         </xs:simpleType>
         <xs:simpleType name="authentication">
            <xs:restriction base="xs:string">
               <xs:enumeration value="Basic"/>
               <xs:enumeration value="SSL"/>
               <xs:enumeration value="Other"/>
            </xs:restriction>
         </xs:simpleType>
      </xs:schema>
```

B. References

1. Charles Lewis and Giuseppe Di Fabbrizio, **A Clarification Algorithm for Spoken Dialogue Systems**, 2005 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP) , Philadelphia, PA, USA, March 18-23, 2005. bibtex

2. Giuseppe Di Fabbrizio and Charles Lewis, **Florence: a Dialogue Manager Framework for Spoken Dialogue Systems**, ICSLP 2004, 8th International Conference on Spoken Language Processing , Jeju, Jeju Island, Korea, October 4-8, 2004. bibtex

3. Giuseppe Di Fabbrizio and Charles Lewis, **An XPath-based Discourse Analysis Module for Spoken Dialogue Systems**, The Thirteenth International World Wide Web Conference , New York NY, May 17-22, 2004. bibtex

4. **Document Object Model (DOM)** http://www.w3.org/DOM/

5. **W3C Multimodal Interaction Framework** http://www.w3.org/TR/mmi-framework/

6. **XML Path Language (XPath)** http://www.w3.org/TR/xpath