Date: April 2008

UML Profile and Metamodel for Voice-based Applications Specification

Version 1.0

OMG Document Number: formal/2008-04-06
Standard document URL: http://www.omg.org/spec/VOICP/1.0/PDF
Associated Files*: http://www.omg.org/spec/VOICP/20070701
http://www.omg.org/spec/VOICP/20070701/voice_metamodel.xml
http://www.omg.org/spec/VOICP/20070701/voice_profile.xml

* original file: ptc/2007-07-04

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org,* under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices

- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

1 Scope

The specification expresses the models using OMG modeling languages. The Voice metamodel is defined as a MOF metamodel. In addition UML is used as one of the concrete syntaxes attached to the metamodel. The specification describes compliance points in “Conformance Points” below. The specification preserves maximum implementation flexibility, no PSM is given to support the specified PIM metamodel. Interoperability and substitutability are guaranteed thanks to the usage of completely defined syntaxes (XMI, UML Profile, and Textual). The degree of support of internalization is Uncategorized, no assumption is made that makes this specification not usable in a specific region.

2 Conformance

Conformance for tools supporting this specification is specified along two orthogonal dimensions: the *syntax dimension* and the *capability dimension*. Each dimension specifies a set of named levels. Each intersection of the levels of the two dimensions specifies a valid conformance point. All conformance points are valid by themselves, which implies that *there is no general notion of* “*Voice conformance*.” Instead, a tool shall state which conformance points it implements, as described below.

2.1 Conformance Points

Any combination of two named levels, one from each dimension, constructs a conformance point. The figure below specifies the 6 different possible conformance points. A tool can claim to be conformant according to one or more of these conformance points.

	Syntax dimension		
Capability	XMI	UML	Textual
Executable			
Exportable			

Figure 2.1 - Conformance Points

By convention a conformance point is denoted using the abbreviation

Voice - <syntax level><capability level>

If a tool complies to various compliance points the following abbreviation can be used:

Voice - <syntax level1><capability level1> - <syntax level2><capability level2>

For example, a tool could be *Voice-XMIExecutable* and *Voice-TextualExportable* and another *Voice-UmlExecutable*. For the first tool the abbreviation *Voice-XMIExecutable-Textual-Exportable* can be used.

2.1.1 Syntax Dimension

The syntax dimension consists of the three named syntax levels:

- XMI: The Voice metamodel serving as the basis for XMI interchange is described in Chapter 8.
- UML: The Voice UML Profile is described in Chapter 9.
- Textual: The textual notation of the Voice language is described in Chapter 10.

2.1.2 Capability Dimension

The capability dimension has two named levels:

- Executable: An implementation shall provide a facility to import or read, and then execute the given syntax (XMI, UML Profile or Textual). The execution shall be according to the semantics of the Voice metamodel.
- Exportable: An implementation shall provide a facility to export a voice dialog definition into one of the three possible syntaxes (XMI, UML Profile or Textual).

3 Normative References

1. Unified Modeling Language (UML), v1.3 specification *(formal/00-03-01)*
2. Meta Object Facility (MOF), v1.3 specification *(formal/00-04-03)*

4 Terms and Definitions

The models and terminology of the UML 1.3, MOF 1.3 and XMI 1.1 specification and the Model Driven Architecture have been used in this specification.

5 Symbols

No specific symbols are defined in this document.

6 Additional Information

6.1 Changes to Adopted OMG Specifications

No changes to the adopted OMG specifications are requested in this specification.

6.2 How to Read this Specification

The rest of this document contains the technical content of this specification. The structure is as follows:

- Chapters 7 (Introduction), 8 (Metamodel), 9 (UML Profile), and 10 (Mapping to VoiceXML) comprise the specification. Annexes A and B contain additional information about the specification.

6.3 Acknowledgements

The following companies submitted and/or supported parts of this specification:

- Alcatel
- EURESCOM
- France Telecom
- IBM
- HP
- Softeam
- Telelogic

A special thanks to Mariano Belaunde (France Telecom) who was the main submitter responsible for preparing this specification.

7 Introduction

7.1 Overview

This specification addresses the need for standardizing a high-level notation for designing *dialogs* in interactive voice response applications, independently of any specific voice-based platform. The VoiceXML specification [VXML] from the W3C defines an executable language for executing audio dialogs.

The VoiceXML specification [VXML] from the W3C defines an executable language for executing audio dialogs. Figure 7.1 shows an example of an interaction described using this language. The language enables a separation of service logic from interaction behavior and frees the developers from resource management. Its major goal is to bring the advantages of web-based development and content delivery to interactive voice response applications. Most of new competing voice portals are based on this standard.

```
C (computer): Would you like coffee, tea or milk?
H (human): Orange juice.
C: I did not understand what you said.
C: Would you like coffee, tea or milk?
H: Tea
C: (continues in document drink2.asp)
```

```
<?xml version="1.0"?>
<vxml version="1.0">
 <form>
      <field name="drink">
      <prompt>Would you like coffee, tea, or milk?</prompt>
          <grammar src="drink.gram" type="application/x-jsgf"/>
      </field>
     <block>
        <submit next="http://www.drink.example/drink2.asp"/>
     </block>
</form>
</vxml>
```

Figure 7.1 - Example of a VoiceXML document

A VoiceXML compliant platform will typically have a multi-tier architecture, as depicted in Figure 7.2. An application server generates dynamically the VoiceXML pages to be executed by the VoiceXML gateway. Distinct voice portal providers may share a VoiceXML gateway to execute the VoiceXML pages. However, for the high-level design of this dialog, there is no standard graphical notation defined: each voice portal provider proposes its own proprietary notation.

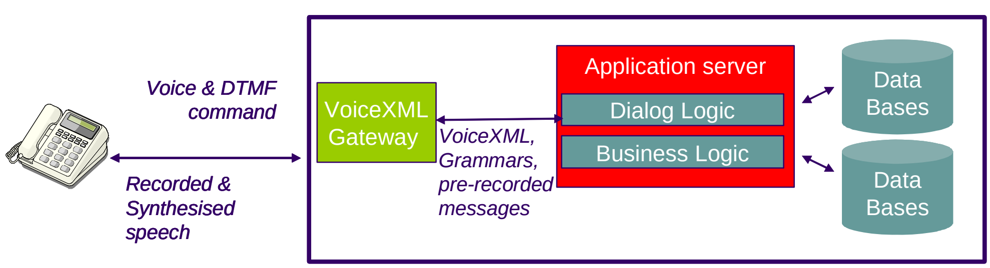


Figure 7.2 - Multi-tier architecture in voiceXML based portals

From an end-user perspective, it is very important to be able to design dialogs independently of the selected voice platform (whether VoiceXML-based or not). Because the technology is rapidly evolving in this field, a voice service provider may need to change the underlying implementation technology and in the meantime re-use the existing dialog specifications.

UML as a well-accepted general-purpose design notation appears as being a natural candidate to serve as the basis for the graphical notation. UML 2 has improved significantly the capacity to describe complex behavior. On the other hand, the MOF formalism has proved to be a convenient way to define the concepts that are relevant to a specific domain, in our case, voice dialog specification.

8 The Voice Metamodel

8.1 Introduction

The Voice metamodel defines the concepts needed to represent complete executable *dialogs*. It contains firstly behavioral concepts that represent the dialog as a *state-machine* – the different kinds of nodes, the transitions – then it contains the concepts to represent the various kinds of input events (DTMF, speech recognition and so on), and finally the concepts to represent basic actions. In addition object oriented structuring (Package, Class, Operation) is used to represent the business code that needs to be manipulated to render the voice service.

Figure 8.1 contains the various packages of the metamodel.

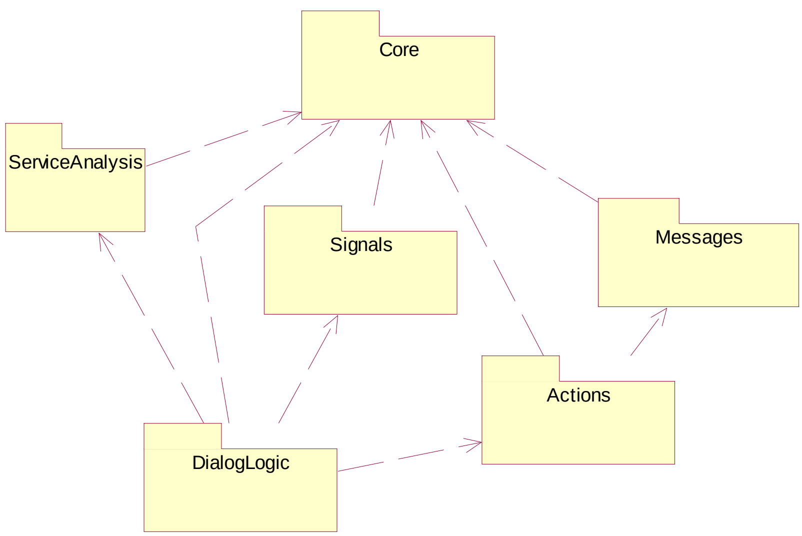


Figure 8.1 - Structure of the Voice metamodel

8.2 Voice Service Modeling

This section presents the concepts needed to describe interactive voice dialogs. In particular this includes the concepts to describe how the dialogs between a voice service and an end-user are sequenced.

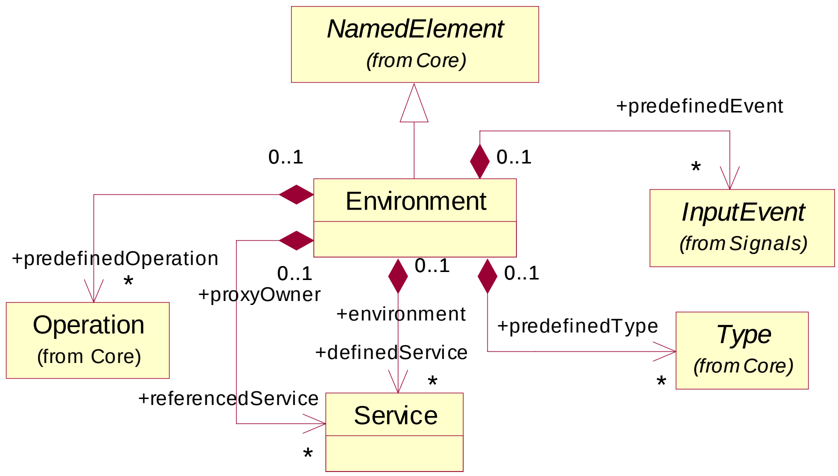


Figure 8.2 - The environment of a voice service

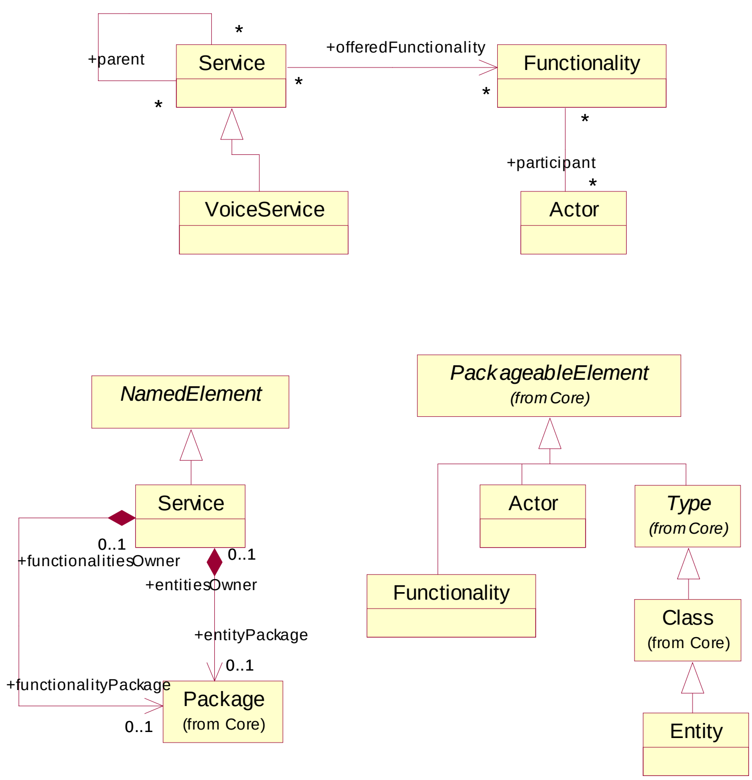


Figure 8.3 - Service specific concepts

8.2.1 Environment

The environment is the root instance for a voice service defining dialogs. It contains all the global declarations used by the dialogs.

8.2.2 Service

A Service represents a coherent set of functionalities that an end-user perceives as a whole and to which it is able to describe. Example: a remote address book hosted by the telecommunication operator and accessed through voice.

Properties

- offeredFunctionality : Functionality

  Designates the list of functionalities offered by this service.

- parent : Service

  The parent service in the hierarchy of declared services.

8.2.3 Entity

An Entity represents any business data that need to be manipulated in order to provide a service. Examples:

- A record containing an entry in the address book of a user.
- An Entity is a kind of Class, which may define properties and operations.

8.2.4 Functionality

A unit of behavior that provides an added-value to the user. A service is decomposed in functionalities. Examples:

- The function that allows consulting its address book.
- The function that allows updating its address book.

8.3 Voice Dialog Modeling

This section presents the concepts needed to describe interactive voice dialogs. In particular this includes the concepts to describe how the dialogs between a voice service and an end-user are sequenced.

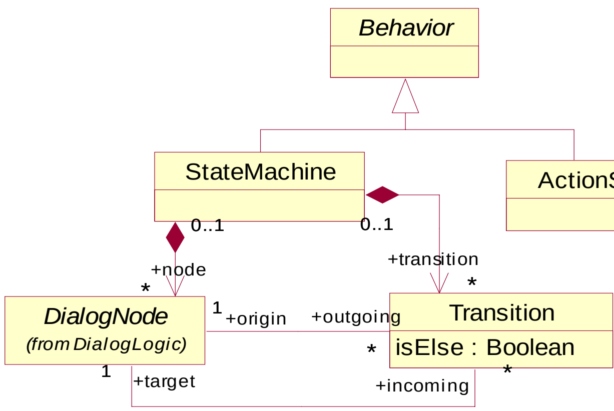


Figure 8.4 - Behaviors

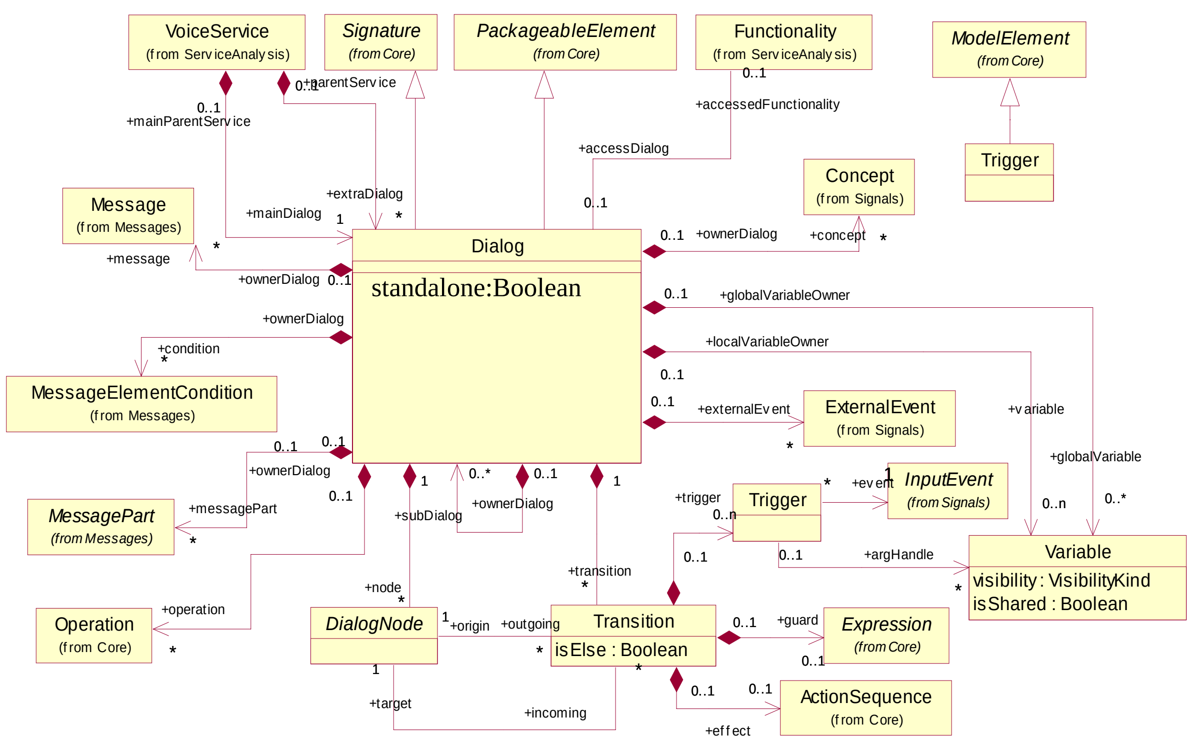


Figure 8.5 - Dialogs

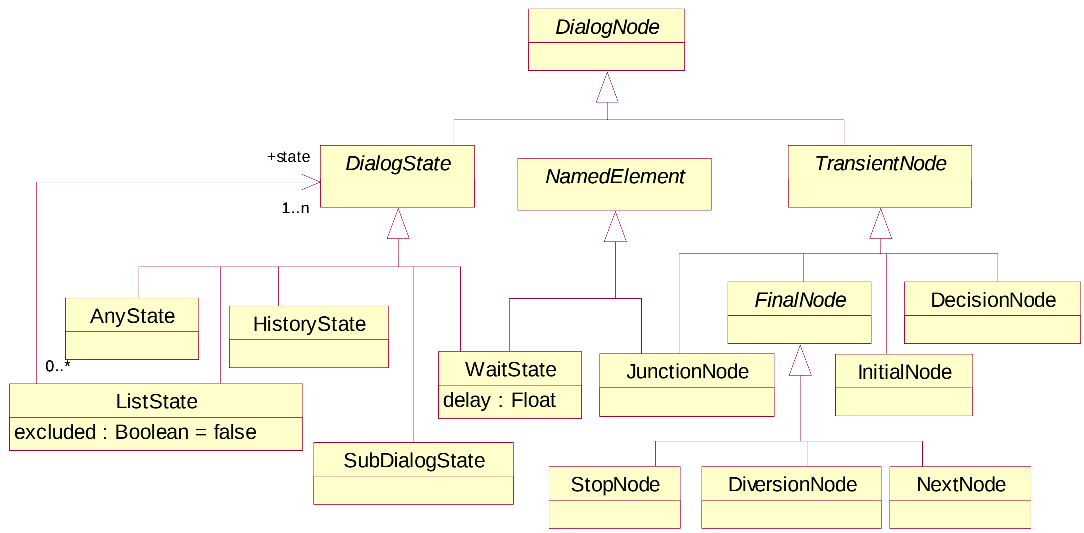


Figure 8.6 - Nodes hierarchy

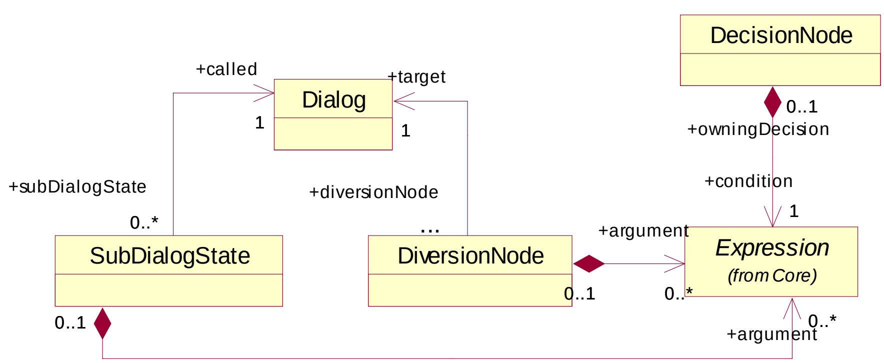


Figure 8.7 - Sub dialogs and diversion nodes

8.3.1 Dialog

A dialog describes the interaction between a voice service and an end-user in order to provide a given functionality. A specific dialog can be associated to the whole voice service. Its purpose is to manage the access to each functionality that is provided by the service.

A dialog is described as a graph of nodes, in which the sequencing of the dialogs is represented thanks to transitions.

Two kinds of nodes play a specific role:

1. The nodes representing the situation where the system waits for a user action (WaitState).
2. The nodes that reference a dialog defined elsewhere (SubDialogState).

These two nodes represent a *stable* situation for the voice system: a state is associated with them. The other nodes are *unstable* nodes or *transient* nodes. The system does not stop when these nodes are reached: they are not states for the system.

A dialog may define and manipulate variables. These variables can contain values to be provided to the user, such as the number of available messages. These variables may also contain data that will influence the flow of dialogs, for instance, the telephone number used by the user when calling the service.

A dialog may have input and output parameters. This is represented by the Signature metaclass, which is a base class of Dialog.

Properties

- accessedFunctionality : Functionality

  The functionalities being used in the dialog.

- globalVariable : Variable

  The variables that are accessible to all dialogs.

  Only the main dialog can declare global variables.

- variable : Variable

  The local variable declared by this dialog.

- concept : Concept

  The concepts that the dialog expects as a result of speech analysis and/or DTMF.

- externalEvent : ExternalEvent

  Events produced by the environment that the dialog is aware. Example: hang-up

- message : Message

  The messages that are defined locally by this dialog.

- condition : MessageElementCondition

  The conditions associated with the conditional parts of a messages owned by this dialog.

- condition : MessagePart

  The message parts used by the messages of this dialog.

- ownerDialog : Dialog

  the owner of the dialog within the hierarchy of dialogs.

  The message parts used by the messages of this dialog.

- operation : Dialog

  Specific reusable behavior defined for this dialog.

- node : DialogNode

  the nodes of the graph representing the behavior.

- transition : Transition

  The transitions of the graph representing the dialog.

- standalone : Boolean

  States whether the dialog is allowed to refer to global variables. If standalone is True, no global variables can be used.

8.3.2 DialogState

A DialogState is an abstraction that represents a situation in which a condition holds (often this condition is implicit). It may represent a passive situation, such as waiting for a user input, or an active situation like executing a sub-dialog.

8.3.3 WaitState

A WaitState represents a situation in which the system expects an action from the user or another kind of event like time expiration or a rejection. It represents a context for the capture or the interpretation of the inputs.

Properties

- delay

  the expiration time parameter before an Inactivity event is generated.

8.3.3.1 SubDialogState

A SubDialogState represents an invocation of a sub-dialog. The sub-dialog is defined separately. When the called sub-dialog terminates its execution, the invoking dialog resumes its execution.

An invocation of a dialog (InvocationDialog) may have arguments (expressions) if the sub-dialog declares parameters.

Properties

- called : DialogState

  The dialog state being invoked.

8.3.3.2 AnyState

When a Transition is associated to an AnyState, this is equivalent to associate the transitions to all the states of the dialog.

8.3.3.3 ListState

When a Transition is associated to a ListState, this is equivalent to associate the transitions to all the states of the list.

8.3.3.4 HistoryState

Represents the state of the dialog that is more recent. It is used to define generic transitions, associated with a list of states or the AnyState. It expressions behaviors like “whatever is the current state, come back after the end of the transition.”

The *deep* property is relevant only if the state to come back is a sub-dialog. When the value is true, the sub-dialog goes to the last visited internal state, and this recursively until reaching a simple state (WaitState). If false, the sub-dialog is re-executed from its default entry point.

8.3.4 Transition

A Transition represents the possibility to go from a node to another node. It represents a control flow between two nodes, that is to say, the set of actions, guards, or event capture that are treated between the two nodes.

From the external environment, an end-user only perceives the stable extremities, that is to say, the nodes where the system pauses and gives the initiative to the user. Between two user actions the system goes from a stable node to another stable node (the nodes that the user can perceive), possibly crossing unstable transitions.

The service exits a stable state by reacting to one of the events that potentially can occur in that state. A typical event will be an action from the user, like a DTMF pressed touch or speaking. Another kind of event is a timer expiration. The system can additionally be simulated by “continuous” signals that are boolean guard conditions. Sometimes it may happen that a transition is triggered only when the two kinds of stimulus occur (a user input or timer expiration and a continuous signal).

The events that are associated with a transition are:

- A *source* node
- An optional *trigger*: corresponds to the presence of non continuous stimuli
- An optional *guard*: a Boolean condition on the data available to the dialog (for instance the current number of inactivities)
- An optional *effect*: the set of actions that are executed if the transition is activated
- The *target* node

Properties

- origin : DialogNode
  The source node of the transition
- target : TargetNode
  The target of the transition
- trigger : Trigger
  A reference of the event to be recognized to execute the transition.
- effect : Action
  The list of actions to execute.
- TransientNode

8.3.5 Transient Node

A TransientNode is an abstraction that represents different kinds of nodes that are not states for the dialog. The different kind of transient nodes are:

- InitialNode: represents the default entry point of the dialog
- ChoiceNode: Represents a conditional branch
- JunctionNode: Denotes a location in the dialog graph to allow redirecting various transitions
- NextNode: End of the dialog and return to the caller

- DiversionNode: End of the dialog with a forced escape to the dialog indicated by the diversion node. The caller ends its execution (no return as for sub-dialogs). Arguments can be passed to the target of the diversion node and it is permitted to invoke recursively the diversion nodes.
- StopNode: Represents the end of the whole service

8.3.6 DialogNode

A dialog node is an abstraction that represents all kinds of nodes that can be a source or a target for a transition.

8.3.7 Trigger

A trigger identifies an event that can produce the activation of a transition. They can be associated with variables, for instance, when the event is the recognition of a word pronounced by the user, this word is stored in an argument of the trigger.

Properties

- event : InputEvent

  The event that is expected to fire the transition.

- guard : expression

  A condition that is required for firing the transition.

8.4 Input Event Concepts

In this section we describe the various kinds of inputs to be managed by the voice service. Figure 8.8 presents this part of the metamodel.

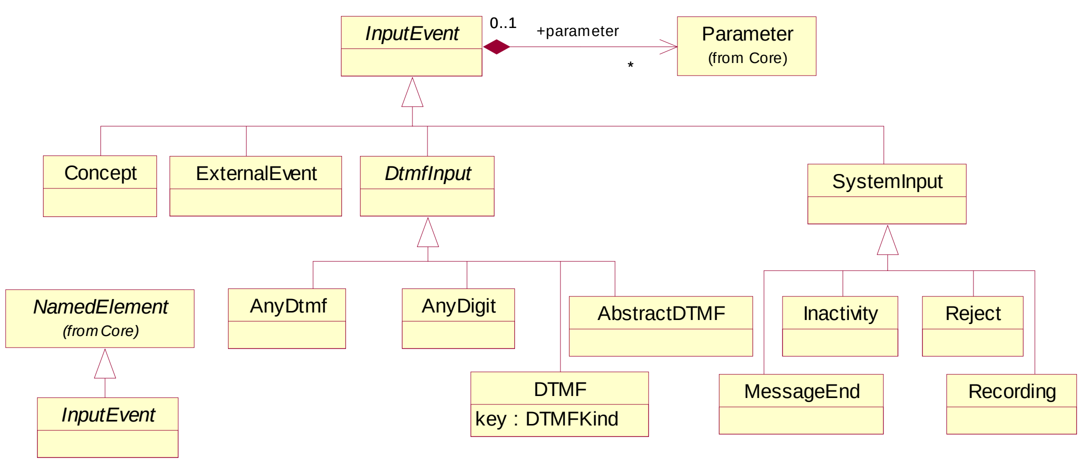


Figure 8.8 - Input events of a dialog

8.4.1 InputEvent

An input event is an abstraction that represents all the kinds of inputs to which a dialog needs to respond.

Properties

- parameter : the slot of the input event used to pass values.

8.4.2 Concept

A Concept is the result of the interpretation of the phrases or words pronounced by the user. This interpretation is produced thanks to speech recognition. If the system uses a semantic analyzer, a Concept typically represents the outcome of the analyzer.

8.4.3 DTMF, AnyDTMF, AnyDigit

Represents a DTMF code. It reflects a press button action from the user on the terminal. The property *key* holds the value of the key being pressed.

AnyDTMF, used in conjunction with a Trigger, represents the arrival of any DTMF code.

AnyDigit, used in conjunction with a Trigger, represents the arrival of any DTMF code, except for the "#" and '*' special characters.

8.4.4 Inactivity

Inactivity represents the fact that the system does not receive any input after a delay expires since a given state of the system is entered. The property *delay* that is associated to any state represents the timeout.

8.4.5 Reject

Reject represents the situation in which the system has detected an input but the confidence on the result is very low.

8.4.6 ExternalEvent

An ExternalEvent represents changes in the environments that potentially affect the dialog, such as the arrival of a message of a change made to a database.

8.4.7 Recording

An event of type Recording represents a phrase or a word pronounced by the user that was not interpreted but stored somewhere for further usage.

8.5 Grammars

Grammars can be explicitly referred in a dialog specification and be attached to signals and to wait states. However the details of the grammar are not defined since this depends on the formalism chosen. The formalism (such as SGRS) and the language (French, English, and so on) can be explicitly indicated.

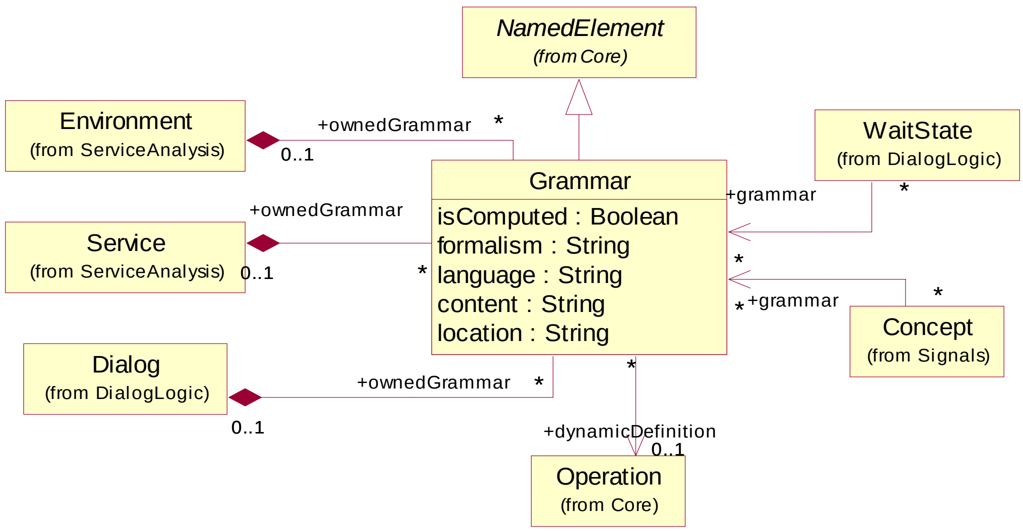


Figure 8.9 - Grammar referencing

8.5.1 Grammar

A Grammar instance represents the usage of a grammar definition in a dialog specification. A grammar can be attached to an utterance signal (Concept), to a WaitState. It can be defined at the level of the environment (top-level), or at the level of a Service, or be specific to a Dialog. A grammar that is automatically computed can have its dynamic definition given as an operation. Alternatively, the content of the grammar may refer to a file (location property) or may be direction included within the grammar instance (through the *content* property).

Properties

- isComputed : Indicates whether the grammar is generated or if it is statically defined.
- formalism : The language being used to specify the grammar.
- content: the formal description of the grammar (when available)
- location : the location where the formal description of the grammar can be found.

8.6 Message Related Concepts

In this section we present how messages are represented in the metamodel.

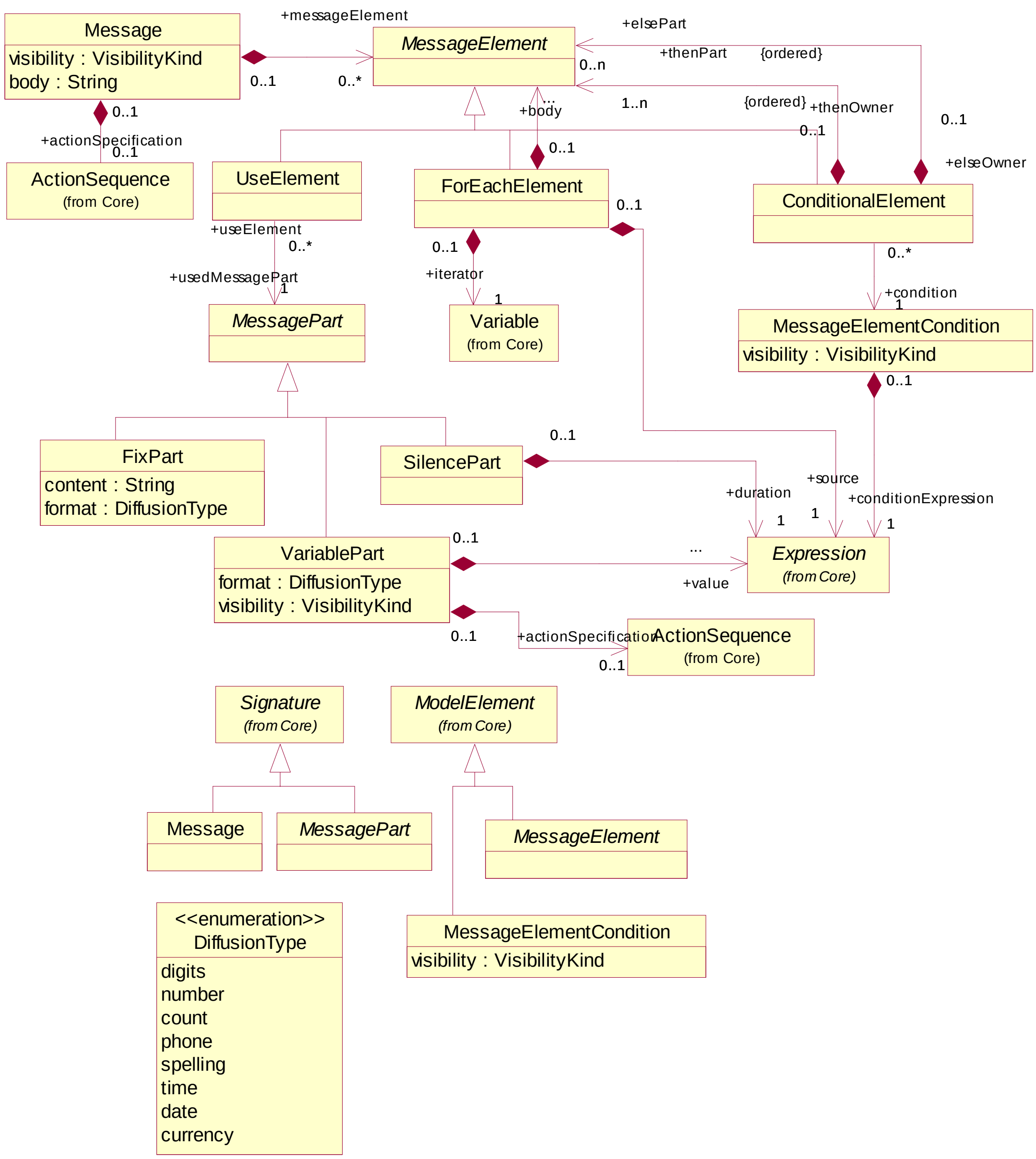


Figure 8.10 - Message structure

8.6.1 Message

A message defines a unit of meaning pronounced by the service (for instance, a phrase). It is composed of a sequence of message elements and may contain conditional parts. It is possible to reuse parts of a message in different messages.

Properties

- messageElement : the parts of this message.
- actionSpecification: an alternative way to specify its content using action.
- visibility : indicates the visibility level of this message within the specification.
- body : a text containing the result of merging the distinct parts.

8.6.2 MessagePart

An abstraction that represents the various elements that are used to build a complete message: fix parts, variable parts, silences.

8.6.3 FixPart

A fix part is a part of a message that is constant and indivisible, which may be recorded or synthesized (text to speech).

Properties

- content : the message to be synthetized and pronounced by the machine.
- format : the format used to render the message.

8.6.4 SilencePart

SilencePart represents a silence that duration is given by an expression.

8.6.5 VariablePart

A variable part represents to the part of a message that results from an expression evaluation. For instance the evaluation of a variable that returns ‘3’ will produce a ‘three’ message part.

Properties

- visibility : indicates the visibility level of this message within the specification.
- format : the format used to render the message.

8.6.6 MessageElement

A MessageElement is an abstraction that represents the different parts of a message (a usage of a message part and the conditional parts).

8.6.7 UseElement

A UseElement represents the usage of a message part within a given message.

8.6.8 ConditionalPart

In a message, some parts may not be pronounced depending on Boolean conditions. If the condition is true, the ‘thenPart’ is pronounced, otherwise the ‘elsePart’ is pronounced.

8.6.9 Condition

A Condition is a Boolean expression that is used as a decision in a conditional message.

8.7 Action Concepts

In this section we describe the kind of actions that can be realized during the execution of the voice dialog.

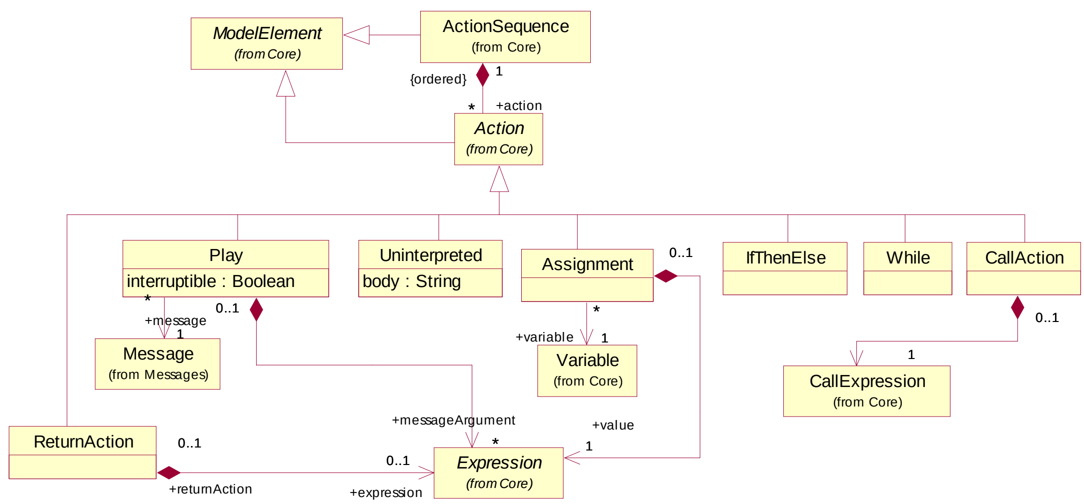


Figure 8.11 - Actions

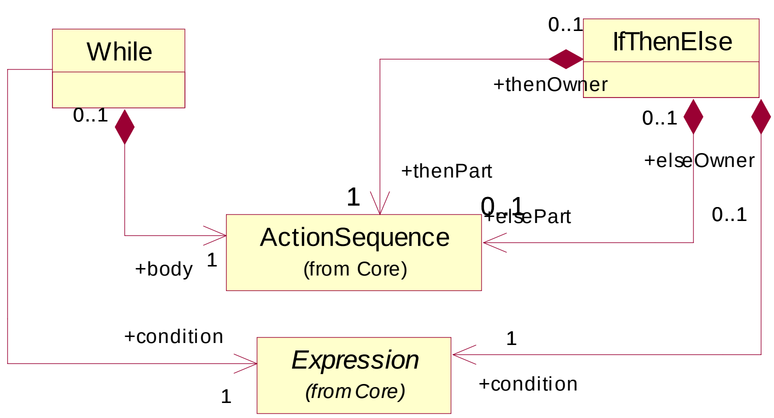


Figure 8.12 - Composite actions

8.7.1 ActionSequence

An action sequence is an ordered list of actions.

8.7.2 Action

An Action is an abstraction that represents the various kinds of actions that can be executed during the provision of a voice service. These actions can be directly called from the dialog node or can be attached to the transitions.

8.7.3 Play

A Play instance represents the action of emitting a message. The play of a message can be interrupted or not interrupted depending on the 'interruptible' property value. If the message has parameters, the action of playing the messages has to provide arguments.

8.7.4 Assignment

This action consists to assign a value to a variable.

8.7.5 Call

This action represents the invocation of an operation, typically an operation hold by a business entity. The call can pass arguments if the called operation declares parameters.

8.7.6 Uninterpreted

An Uninterpreted instance represents an action described informally (typically using natural language).

8.7.7 Return

This action represents the return of an operation.

8.7.8 IfThenElse

This action represents a conditional action.

8.7.9 While

This action represents a loop that will stop when the related condition evaluates to false.

8.8 Core Concepts

In this section we describe the structuring concepts needed to represent business data and business code. The concepts are mainly taken from UML 2 and MOF 2. The expressions are used in guards and in actions.

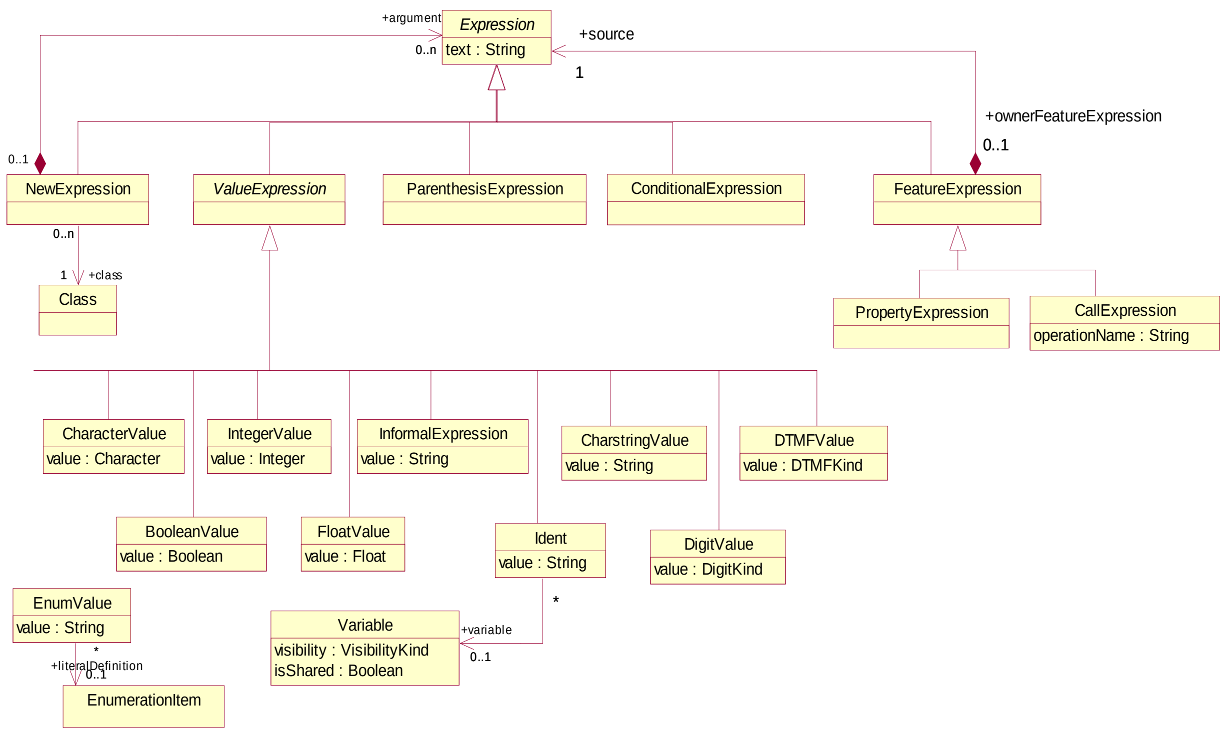


Figure 8.13 - Expressions

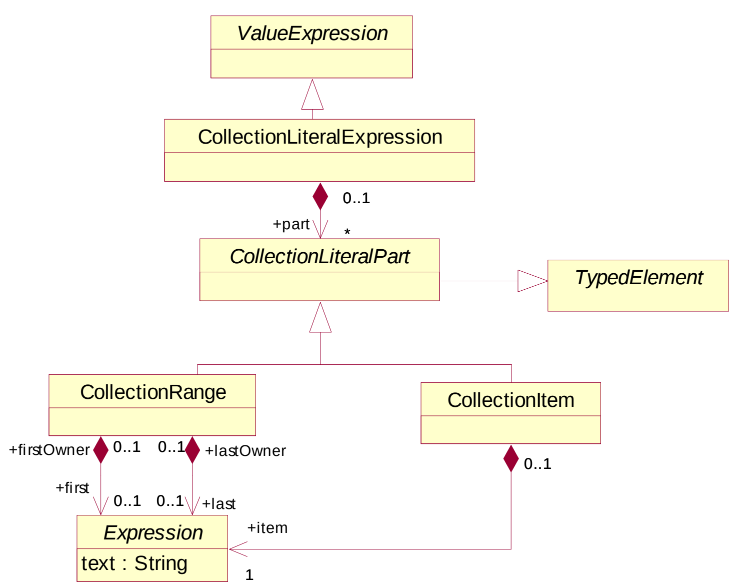


Figure 8.14 - Literals

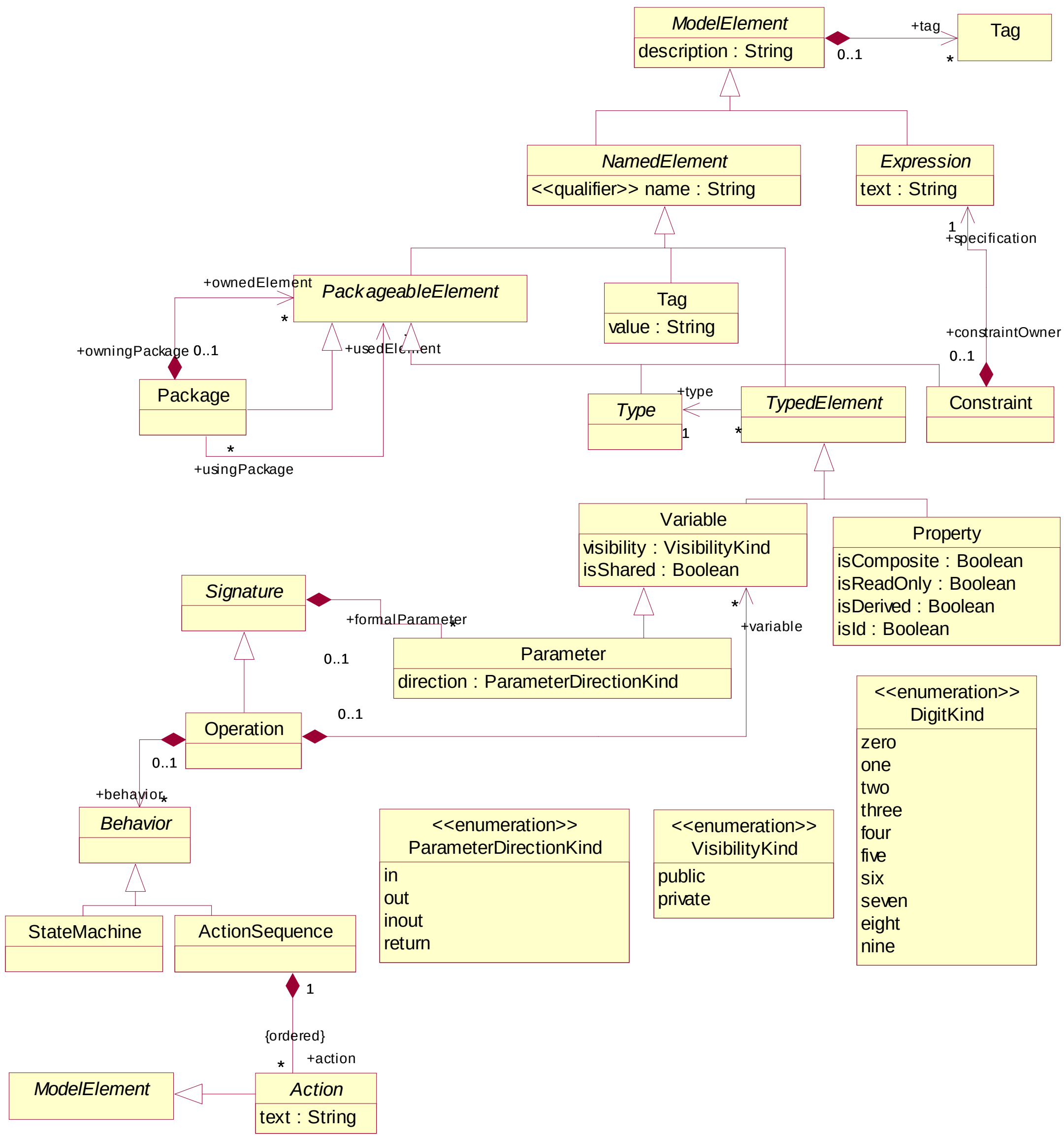


Figure 8.15 - Core structuring concepts

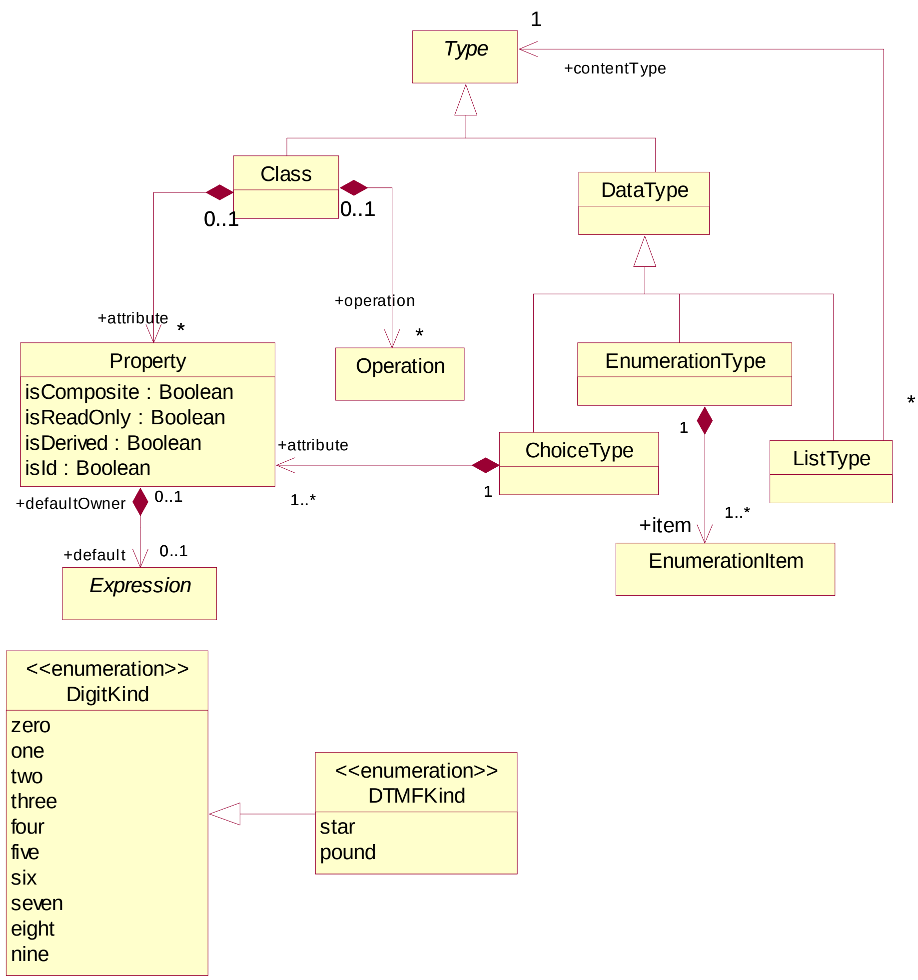


Figure 8.16 - Types

9 The Voice UML Profile

In this chapter we describe the UML 2.0 profile associated with the Voice metamodel described in Chapter 8.

The profile is described using:

- a table that gives for each "voice" concept the corresponding UML concepts and the graphical representation.
- a table with the list of all defined stereotypes, the base classes, and the tagged values associated with these stereotypes.

Some examples are provided to illustrate the usage of the UML notation.

9.1 Structure of a Voice Service Model

A UML model may contain the definition of a single voice service or the definition of various voice services. A "Framework" package contains the lists of predefined signals and predefined operations that are available to all services. Each voice service is represented by a Package stereotyped <<Service>>. A package containing the definition of entities can be either contained within a <<Service>> package or live at same level - typically imported from other UML models. The latter is useful for services sharing the same set of entities.

The structure of a <<Service>> package should follow one of the two structural schemes:

Old style:

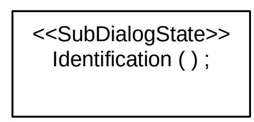
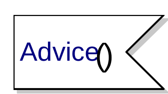
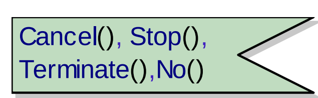
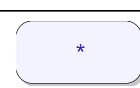
Entities defined specifically for the service are defined within a package stereotyped <<EntitiesModel>>. The dialogs are represented by a package stereotyped <<DialogModel>>. The main dialog is the unique <<DialogModel>> package directly contained by the <<Service>> package. A <<DialogModel>> has the following structure: a class stereotyped <<InputContainer>> to contain the locally defined input events (UML signals), a class stereotyped <<VariableContainer>> to contain the global variables (only for the main dialog), a class stereotyped <<MessageContainer>> to contain the messages (UML operations), and a class stereotyped <<BehaviorContainer>> to contain the operation containing the behavior definition (state machine or activity graph). Sub dialogs are defined by nested packages stereotyped <<DialogModel>>.

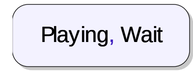
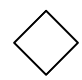
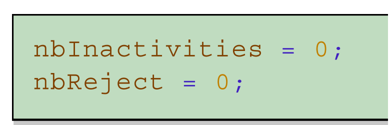
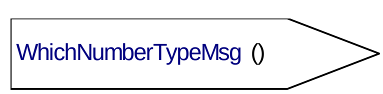
New style:

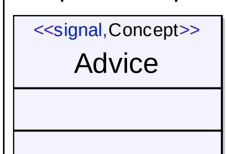
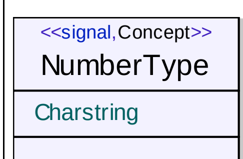
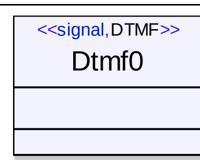
Within the <<Service>> package, a dialog is directly defined by a behavior (either a state machine or an activity graph). The main dialog is the unique behavior directly contained by the <<Service>> package. Sub dialogs are defined as behaviors owned by the behavior representing the owning dialog. The input events are defined as signals owned by the <<Service>> package. Variables are defined as properties of the behavior and messages are defined as operations of the behavior.

These two styles are needed to cope with existing UML implementations. Old style can be used by UML 1.x conformant tools or UML2 tools that do not support the ability for a behavior to contain properties and operations.

9.2 Voice Metamodel to UML Correspondences

Voice Metamodel Concept	UML 2.0 Concept	Notation
VOICE DIALOGS		
Dialog	State machine stereotyped <<Dialog>>	One or more state-transition diagrams
WaitState	State stereotyped <<WaitState>>	
SubDialog-State	Action stereotyped <<SubDialogState>>	
Transition	Transition.	Transition arrow. The trigger and the actions of the "whole" transition are explicitly drawn as nodes linked by transitions.
Trigger	Trigger	A unique trigger symbol  Multiple triggers 
Guard	Constraint	Within a transition within a trigger: expression with brackets attached to the transition arrow.
AnyState	State named "*"	

ListState	ListState	
Transient Node	Pseudostate	Specific to each kind of pseudostate.
InitialNode	Pseudostate with kind Initial	
ReturnNode	FinalState	
DiversionNode	FinalState stereotyped "diversion"	
ChoiceNode	Choice	
HistoryNode	DeepHistory or ShallowHistory	 (for *deep*)
JunctionNode	Junction	 name
ACTIONS		
Action Sequence	Activity	Rectangle containing the list of actions.  Alternative: sequence of rectangles connected by transition arrows. **Note***:* The action of playing a message is represented differently through the usage of a send symbol (see Play).
Play	SendSignal-Action	

Assignment	WriteStructural-FeatureAction WriteVariable-Action	Specific keywords using a Java like notation. **Note**: UML 2.0 does not define a concrete syntax for the specific actions.
Uninterpreted	Comment	
Return	ReturnAction	*return* keyword
IfThenElse	ConditionalNode	*If then else* keywords
While	LoopNode	*while* keyword
INPUT EVENTS		
InputEvent	Signal	
Concept	Signal stereotyped <<Concept>>	Simple concept:  Parameterized concepts: 
DTMF	Signal stereotyped <DTMF>>	

ExternalEvent	Signal stereotyped <<External Event>>	
Voice metamodel Concept	**UML concept**	**Textual Representation**
MESSAGES		

Message	An operation stereotyped <<Message>> with a return parameter of type String. The operation returns the concatenation of the message parts.	public static <<Message>> Charstring M_1 {
		return (cond_1?
		(FP_2()):
		(FP_3())+FP_3());
		}

FixPart	Operation stereotyped <<FixPart>> With a tagged value 'format', which default value indicates the format of the string (a date, a phone number, and so on).
	The operation returns a string that represents the fix part to be pronounced.
		Public static <<FixPart>> Charstring FP_1 ()
		{
		return "Bonjour";
		}

Silence	Operation stereotyped <<Silence>>. The operation returns a string that is the result of a call to a pre-defined "Silence" operation with a parameter to pass the duration of the silence.	Public static<<SilencePart>> Charstring S_1 () {
		Silence (3);
		}

VariablePart	Operation stereotyped <<VariablePart>>: With a tagged value 'format,' which default value indicates the format of the string (a date, a phone number, and so on). The operation has a return value of string type and returns the evaluation of an expression that provides the content of the variable part.	Public static <<VariablePart>> VP_1 () { nom; }
Condition	Operation stereotyped <<Conditional>> The operation has a return parameter of type boolean and its body is a boolean expression.	public static <<ConditionPart>> Boolean C_1 { return (heure>17)}

UseElement	A call to the operation that represents the part of the message that is used. This invocation should be done in the body of the operation that represents the whole message.	public static <<Message>> Charstring M_1 { return (cond_1?(**FP_2()**):(**FP_3()**)+**FP_3()**);}
Conditional Element	Conditional expression within the body of the operation representing the message.	public static <<Message>> Charstring M_1 { return (**cond_1?(FP_2()):(FP_3())**+FP_3());}

9.3 Stereotypes of the UML Voice Profile

In this section we provide the list of stereotypes and, when applicable, the list of tagged values associated to a specific stereotype. No specific icons are defined to represent these stereotypes. In general the name of the stereotype corresponds with the name of the underlying concept in the Voice metamodel.

Stereotype	UML 2.0 Base class	Voice MM concept	Tagged Values
<<Dialog>>	StateMachine	Dialog	
<< WaitState >>	State	WaitState	
<<SubDialogState>>	Action	SubDialogState	
<<Diversion>>	FinalState	DiversionNode	
<<Accept>>	Trigger	Trigger	
<<Concept>>	Signal	Concept	
<<DTMF>>	Signal	DTMF	
<<ExternalEvent>>	Signal	ExternalEvent	
<<MessageContainer>>	Class	Ownership of Message	
<<Inputcontainer>>	Class	Ownership of InputEvent	
<<Message>>	Operation	Message	
<<Silence>>	Operation	Silence	
<<FixPart>>	Operation	FixPart	format : String
<<VariablePart>>	Operation	Variable	format : String
<<Service>>	Package	Service Root of the definitions for a given VoiceService	

Stereotype	UML 2.0 Base class	Voice MM concept	Tagged Values
<<DialogModel>>	Package	Ownership of all the packages used to define the dialogs of a given service. Should be contained by a Service package.	
<<EntitiesModel>>	Package	Ownership of the packages defining or declaring the business entities accessed by the voice service.	
<<BehaviorContainer>>	Class	Ownership of the Operation containing the State Machien representing the behavior.	
<<Dialog>>	Operation	Dialog. Ownership of the parameters of the Dialog and of the state machine defining the interaction.	

The following stereotypes are only applicable when the “old style” structural schema (see Section 9.1) is used: <<MessageContainer>>, <<InputContainer>>, <<BehaviorContainer>>, and <<DialogModel>>.

9.4 Using Activity Diagrams to Represent Dialog Behavior

For more flexibility in the implementation, the dialog behavior which, from a semantic point of view is defined by a state machine, can be rendered by an activity diagram following some conventions.

When this variation in notation is used the following mappings should apply:

- An ActivityGraph replaces a StateMachine.
- An InitialNode replaces a Pseudostate with kind=initial
- A DecisionNode replaces a Pseudostate with kind=choice
- A MergeNode replaces a Pseudostate with kind=junction
- An ActivityFinalNode replaces a FinalState
- An Action replaces a State

The base classes for the stereotypes are changed according to these mappings.

9.5 Examples

This section presents some examples to illustrate the usage of the UML notation to modelize voice dialogs that are compliant with the metamodel defined in Chapter 8. These examples are taken from France Telecom voice services.

9.5.1 A Main Identification Dialog

The dialog depicted by Figure 9.1 shows a dialog performs pronounces a welcome message and then performs an identification of the user. At the end of this step a parameter is returned indicating if the identification succeeded. If the identification is OK, the dialog is branched (DiversionNode) to another dialog; otherwise a warning message is pronounced and then the connection is closed.

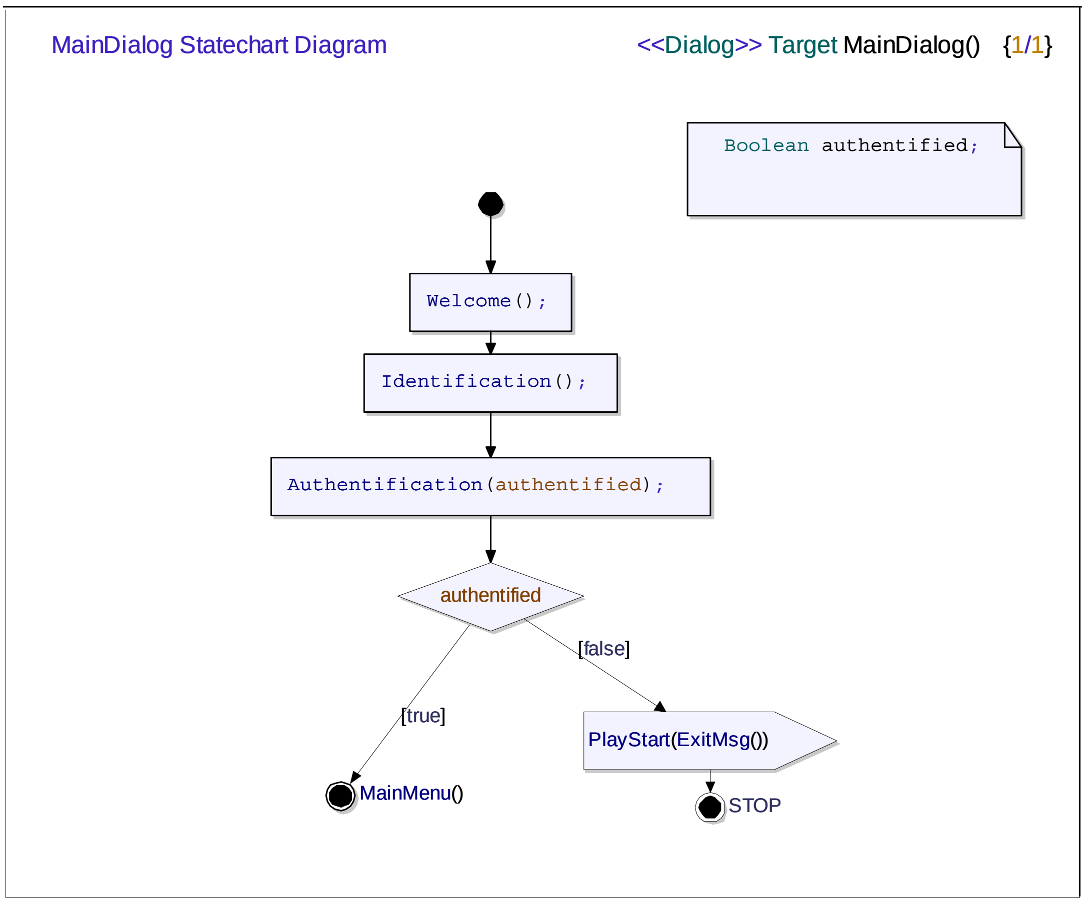


Figure 9.1 - An identification dialog

9.5.2 A Dialog to Check Feasibility

In this example the dialog checks if the service that is requested can be provided. This dialog will typically be reused by different services. This dialog makes a call to a business entity (named PARSI in the figure) in order to decide what to do. He delivers a non-interruptible message (modeled as PlayAll instead of PlayStart), he assigns the result, and then terminates giving the control to the caller (final symbol named NEXT).

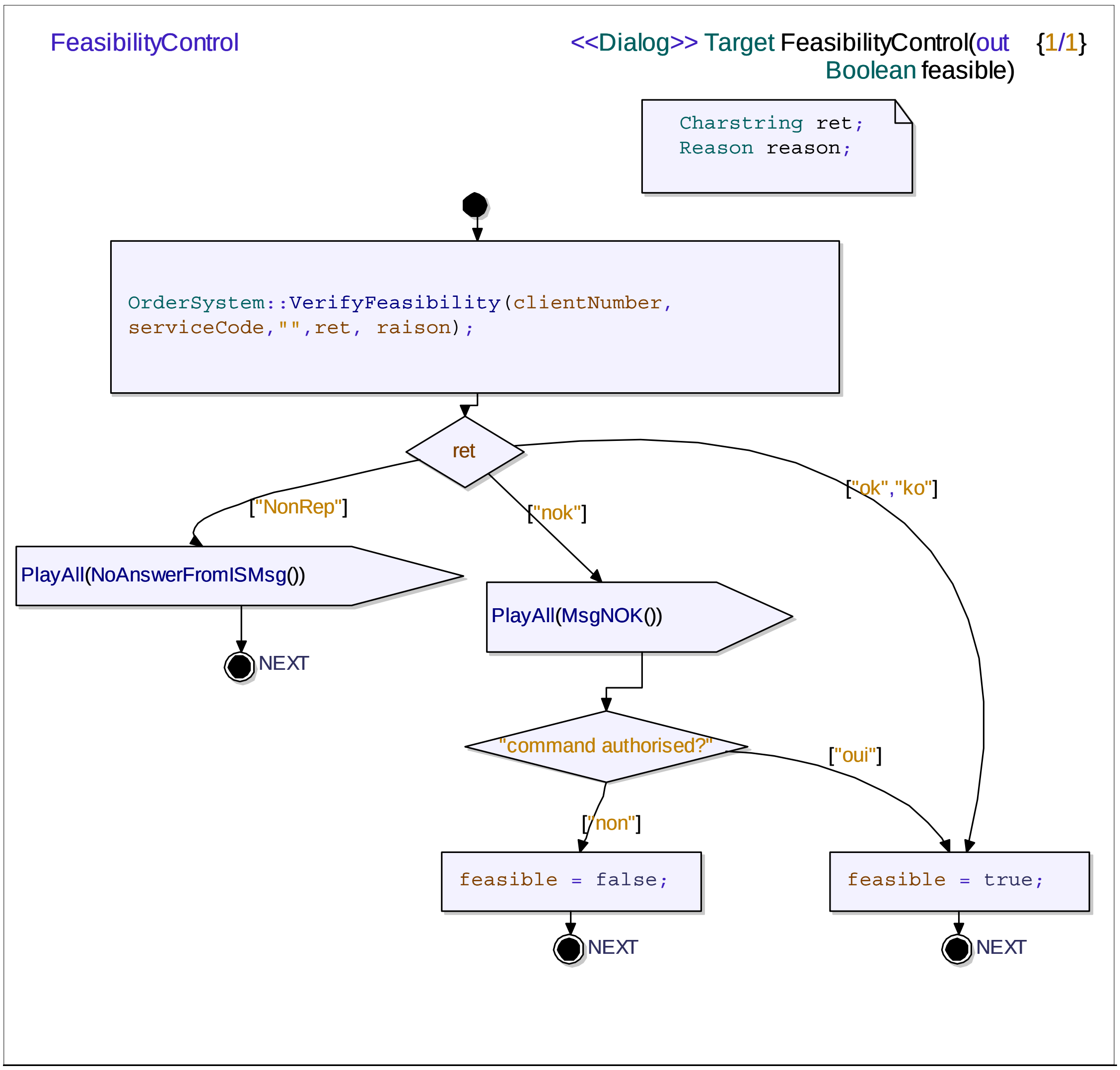


Figure 9.2 - A reusable feasibility check dialog

9.5.3 A Menu Dialog

This example illustrates a kind of menu dialog that asks the user if he wants to order something or just retrieve some information from the system. The machine waits for an input of the user, which can be a DTMF key or a phrase pronounced by the user.

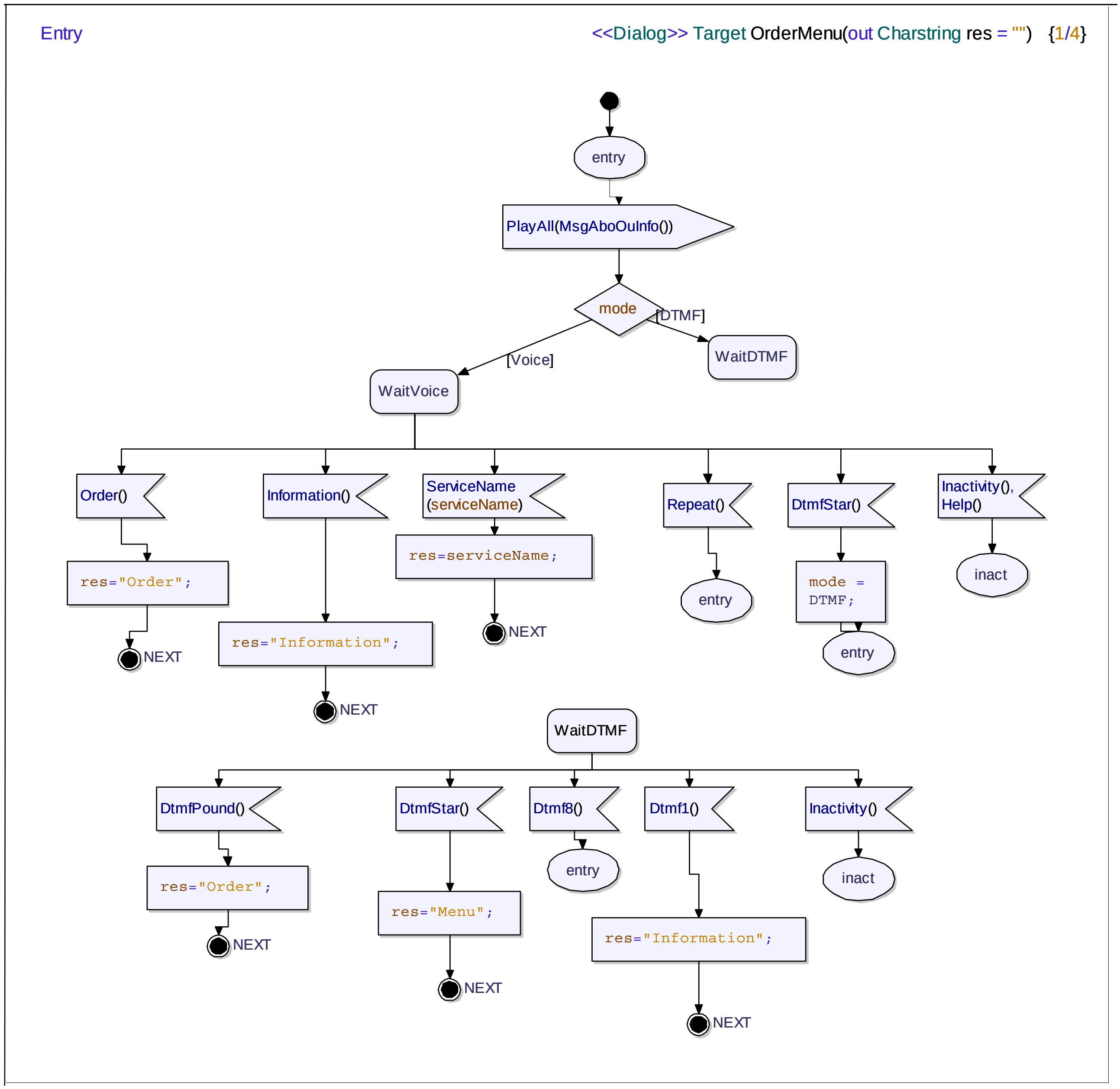


Figure 9.3 - A menu dialog

For each wait state there are transitions that describe the reaction of the services to the user stimuli. Each transition starts with a trigger, which may be a concept or a DTMF key, or an inactivity from the user. A transition can have a set of triggers, meaning that it can be activated by any of the triggers.

In the dialog description it is possible to do a branch to a given point of the dialog. This is expressed thanks to a junction node.

10 Textual Notation

In this chapter we define a textual notation associated with the Voice metamodel. This notation is useful to voice dialog designers that have a “programmer” background. It is also useful to implement the Voice profile more easily since the details of a dialog – such as the actions and the body of the messages can be provided textually. Hence the UML tool implementing the profile will not be required to provide a complete support of this detailed part.

10.1 Examples

This section illustrates the usage of the notation. The identification dialog in Figure 9.1 can be rendered textually using the following syntax:

```
dialog identification {
 message ExitMessage() {return "Good bye";}
 behavior() {
  var auth:Boolean;
  call Welcome():
  call Identification();
   call Authentification(auth);
   decision {
      case "true" { divert mainDialog();}
       case "false" {plays}
    }
   stop;
   } // end of dialog behavior
} // end of dialog
```

10.2 Grammar of the Concrete Syntax

This section gives formally the grammar.

Lexical elements:

The list of reserved words:

service	voiceservice	entities	package	class	operation	message
messagepart	event	externalevent	systemevent	static	global	shared
property	var	extends	maindialog	dialog	within	in
inout	out	behavior	play	playall	call	divert
return	stop	decision	case	junction	jump	restart
wait	when	do	accept	if	then	else
endif	null	true	false	unlimited	not	and
or	xor	informal	new	Set	Bag	Sequence
OrderedSet	standalone					

In the BNF these keywords are denoted by the corresponding word in capital letters. For instance DIALOG denotes the occurrence of the dialog keyword.

The following variable tokens are defined:

ID: an alphanumeric identifier

ICONST: integer value

FCONST: float value

SCONST: double quoted string

CCONST: single quoted string

The following character tokens are defined:

'PLUS' -> '+'

'MINUS' -> '-'

'TIMES' -> '*'

'DIVIDE' -> '/'

'MOD' -> '%'

'EQ' -> '=='

'LT' -> '<'

'LE' -> '<='

'LT' -> '<'

'GE' -> '>='

'GT' -> '>'

'NE' -> '<>'

'NEX' -> '!='

'EQUALS' -> '='

'PLUSEQUAL' -> '+='

'MINUSEQUAL' -> '-='

'ARROW' -> '->'

'PERIOD' -> '.'

'LPAREN' -> '('

'RPAREN' -> ')'

```
'LBRACKET' -> '['

'RBRACKET' -> ']'

'LBRACE'   -> '{'

'RBRACE'   -> '}'

'COMMA'    -> ','

'SEMI'     -> ';'

'COLON'    -> ':'

'DCOLON'   -> '::'
```

BNF

```
toplevel : module_definition_list_opt
module_definition_list_opt : module_definition_list
                    | empty
module_definition_list : module_definition
        | module_definition_list module_definition
module_definition : service
                    | entities
                    | dialog
service : service_kind ID SEMI
service_kind : SERVICE
                    | VOICESERVICE
entities : entities_indicator package_def
entities_indicator : ENTITIES
package_def : package_head LBRACE package_content_list_opt RBRACE
package_head : PACKAGE ID
                    | PACKAGE
package_content_list_opt : package_content_list
                    | empty
package_content_list : class
               | package_def
               | package_content_list class
               | package_content_list package_def
class : class_def
               | class_decl

class_def : class_head LBRACE class_content_list_opt RBRACE
class_decl : class_head SEMI
class_head : CLASS ID class_extension_opt
  class_content_list_opt : class_content_list
               | empty
class_content_list : property
               | operation
               | class_content_list property
               | class_content_list operation
property : property_kind_list declarator SEMI
property_kind_list : property_kind
```

```
               | property_kind_list property_kind
property_kind : PROPERTY
               | VAR
               | SHARED
               | STATIC
               | GLOBAL
property_list : property
               | property_list property
id_list : ID
              | id_list COMMA ID
simple_signature : LPAREN param_list_opt RPAREN
signature : simple_signature
               | simple_signature COLON param_list
  param_list_opt : param_list
               | empty
param_list : param
               | param_list COMMA param

param : declarator
               | param_direction declarator
param_direction : IN
               | INOUT
               | OUT
simple_declarator : type_specifier
               | ID COLON type_specifier
declarator : simple_declarator
               | simple_declarator EQUALS expr
operation : operation_decl
               | operation_def
operation_decl : operation_header SEMI
operation_def : operation_header LBRACE operation_body RBRACE
operation_header : operation_kind ID signature
operation_kind : OPERATION
                | MESSAGE
                | MESSAGEPART
                | EVENT
                | EXTERNALEVENT
                | SYSTEMEVENT
operation_body : action_list_opt
class_extension_opt : class_extension
                | empty
class_extension : EXTENDS scoped_id
scoped_id : ID
                | scoped_id DCOLON ID
type_specifier : scoped_id
                | type_constructor LPAREN type_specifier RPAREN
dialog : dialog_decl
                | dialog_def
'dialog_decl : dialog_head SEMI'
'dialog_def : dialog_head LBRACE dialog_content_list_opt RBRACE'
'dialog_head : ’standalone’? dialog_kind ID within_dialog_opt'
```

```
within_dialog_opt : within_dialog
              | empty
within_dialog : WITHIN ID
dialog_kind : MAINDIALOG
             | DIALOG
dialog_content_list_opt : dialog_content_list
             | empty
dialog_content_list : dialog_content
             | dialog_content_list dialog_content
dialog_content : dialog_behavior
             | property
             | operation
dialog_behavior :  dialog_behavior_head LBRACE behavior_content RBRACE
dialog_behavior_head :  BEHAVIOR simple_signature
behavior_content : property_list node_list_opt
              | node_list_opt
node_list_opt : node_list
             | empty
node_list : node
             | node_list node
simple_node_list : simple_node
             | simple_node_list simple_node
node : simple_node
             | complex_node
simple_node : prompt
             | subdialog
             | control
             | do
complex_node : decision
             | wait
             | when
prompt : PLAY expr SEMI
             | PLAYALL expr SEMI
subdialog : diagcallkind expr SEMI
diagcallkind : CALL
             | DIVERT
control : RETURN SEMI
             | JUMP ID SEMI
             | JUMP jump_kind COLON ID SEMI
             | JUNCTION ID SEMI
             | RESTART SEMI
             | STOP SEMI
jump_kind : WAIT
             | JUNCTION
             | DECISION
decision : decision_head LBRACE decision_body? RBRACE
decision_head : DECISION ID LPAREN expr RPAREN
              | DECISION LPAREN expr RPAREN
wait : wait_head LBRACE wait_body RBRACE
wait_head : WAIT ID
when : WHEN expr node
```

```
do : do_head LBRACE action_list_opt RBRACE
        | do_head action
do_head : DO
arg_list_opt : arg_list
        | empty
arg_list : expr
        | arg_list COMMA expr
unary_op : MINUS
        | NOT
        | INFORMAL
        | NEW

access_op : PERIOD
        | ARROW
logic_and_op : AND
        | XOR
cmp_op : EQ
        | NE
        | LT
        | GT
        | LE
        | GE
add_op : PLUS
        | MINUS
mult_op : TIMES
        | DIVIDE
        | MOD
expr : or_expr
or_expr : and_expr
        | or_expr logic_or_op and_expr
and_expr : cmp_expr
        | and_expr logic_and_op cmp_expr
cmp_expr : additive_expr
        | cmp_expr cmp_op additive_expr
additive_expr : mult_expr
        | additive_expr add_op mult_expr

mult_expr : unary_expr
        | mult_expr mult_op unary_expr
unary_expr : postfix_expr
        | unary_op unary_expr

postfix_expr : primary_expr
        | postfix_expr LBRACKET expr RBRACKET
        | postfix_expr LPAREN arg_list_opt RPAREN
        | postfix_expr access_op ID


primary_expr : literal
        | scoped_id
        | LPAREN expr RPAREN
```

```
literal : literal_simple
        | literal_collection
literal_collection : type_constructor LBRACE collection_item_list_opt RBRACE

collection_item_list_opt : collection_item_list
        | empty

collection_item_list : expr
        | collection_item_list COMMA expr

type_constructor : SET
        | BAG
        | SEQUENCE
        | ORDEREDSET

literal_simple : ICONST
        | FCONST
        | CCONST
        | SCONST
        | TRUE
        | FALSE
        | UNLIMITED
        | NULL
action_list_opt : action_list
        | empty
action_list : action
        | action_list action
action : expr SEMI
        | expr EQUALS expr SEMI
        | IF expr THEN action ENDIF
        | IF expr THEN action ELSE action ENDIF
        | RETURN expr SEMI

decision_body : case_element
        | decision_body case_element
'case_element : case_head LBRACE case_body RBRACE'

case_head : CASE expr
        | ELSE

case_body : simple_node_list
        | empty

wait_body : trigger_element
        | wait_body trigger_element

trigger_element : trigger_head LBRACE trigger_body RBRACE
trigger_head : ACCEPT event_call_list
        | ACCEPT LBRACKET expr RBRACKET event_call_list
        | ELSE
event_call_list : expr
```

```
            | event_call_list COMMA expr
trigger_body : simple_node_list
            | empty
```

Annex A: General Requirements

(informative)

A.1 Summary Of Requests Versus Requirements

The conformance points defined by this specification (see Section 2.1) allow a tool to support only one of the three input syntaxes associated to the Voice metamodel (XMI serialization, UML Profile, or Textual).

A.2 Resolution Of General Requirements

The specification follows the general requirements of the RFP. We provide here a summary of how these general requirements are resolved.

The specification expresses the models using OMG modeling languages: The Voice metamodel is defined as a MOF metamodel. In addition UML is used as one of the concrete syntaxes attached to the metamodel. The document specifies conformance points in Section 2.1. The document preserves maximum implementation flexibility: no PSM is given to support the specified PIM metamodel. Interoperability and substitutability is guaranteed thanks to the usage of completely defined syntaxes (XMI, UML Profile, and Textual). The degree of support of internalization is Uncategorized: no assumption is made that makes this specification not usable in a specific region.

Annex B: References

(Informative)

1. Metamodel and a UML profile for Voice Applications RFP, OMG Document telecom/2004-04-02
2. Voice XML Markup language http://www.w3.org/TR/2003/CR-voicexml20-20030220/
3. Speech Recognition Grammar Specification

   http://www.w3.org/TR/2002/CR-sppech-grammar-20020626/
4. IST MODA-TEL project: MDA applied to telecommunications: http://www.modatel.org
5. How to build a speech recognition application, Bruce Balentine and David Morgan. EIG Press.

INDEX