Technical Report

ECMA TR/88

1st Edition / June 2004

Designing an Object Model for ECMA-269 (CSTA)

Technical Report

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

Technical Report
ECMA TR/88

1st Edition / June 2004

Designing an Object Model for ECMA-269 (CSTA)

Brief History

Although ECMA-269 (CSTA) specifies its basic objects, CSTA functionality is largely available through a service model, e.g., via XML schemata in ECMA-323 and through Web Services in ECMA-348. Adoption of CSTA for interactive voice services has resulted in increased demand for object-oriented access to CSTA. This Technical Report demonstrates how CSTA objects and behaviour, in particular for use with interactive voice services, can be modelled under the principle of ECMA-335 (CLI). This TR illustrates examples in ECMA-334 (C#), a language member of the CLI family.

This Ecma Technical Report has been adopted by the General Assembly of June 2004.

Table of contents

1 Scope 1

2 References 1

3 Brief Review of CSTA Operational Model 1

4 Key Concepts of Service and Object Models 2

5 CSTA Object Model 3

5.1 CSTA Objects 3

5.2 Services and Events 3

5.3 Service Acknowledgements 5

6 Illustrative Call Control Examples 5

6.1 Starting a CSTA application with implicit association 5

6.2 Placing a monitor on a device 6

6.3 Notification of an inbound alerting connection 7

6.4 Answering an inbound alerting connection 9

6.5 Clearing a connection 9

6.5.1 Service Request 9

6.5.2 Notification of a cleared connection 10

6.6 Initiating an outbound call 10

6.6.1 Service Request 11

6.6.2 Outbound Call Event Sequence 11

6.7 Single step transfer or Deflect call 13

6.7.1 Service Requests 13

6.7.2 Notification events 14

6.8 Single step conference or Join Call 15

6.8.1 Service Requests 16

6.8.2 Notification Event 16

7 Illustrative Examples for Interactive Voice Functions 18

7.1 Obtaining an object 18

7.2 Attaching the audio 18

7.3 Simple prompt playback 19

7.4 Advanced prompt playback 19

7.4.1 Set Voice Attribute for the filler prompt 20

7.4.2 Play the filler prompt 20

7.4.3 Prepare the new prompt 21

7.4.4 Stop the filler prompt 21

7.5 Simple speech recognition 21

7.5.1 Obtain device and attach audio 22

7.5.2 Set up recognition grammars 22

7.5.3 Attach event handlers and start recognition 22

7.6 Simultaneous DTMF and speech recognition 24

7.7 Simultaneous speech recognition and speaker verification 24

7.8 Sharing voice devices 25

Annex A (informative) - A comparison between CSTA and SALT 1.0 27

1 Scope

With the introduction of TR/85, “Using ECMA-323 (CSTA XML) in a Voice Browser Environment,” CSTA has witnessed a strong adoption in the area of interactive voice services. Software agents, equipped with speech recognition and synthesis capabilities, are deployed in call centres to provide automated services. Leveraging the rich functionality of CSTA, businesses are able to offer customers around the clock services without being limited to office hours or personnel constraints.

Accompanied with the strong adoption is the demand for simpler access to CSTA functionality. One such demand for CSTA concerns the need of CSTA in an object oriented programming style, the mainstream computer software development paradigm. Although CSTA has been specified in a manner consistent with an object oriented design, the CSTA Standard Suite has been exclusively composed of specifications that make CSTA functionality available in a service model. ECMA-323 and ECMA-348 specify an XML based syntax and WSDL for CSTA respectively.

This Technical Report demonstrates how developers can use CSTA in an object-oriented fashion. To broaden the reach, this TR bases the discussion on ECMA-335 (Common Language Infrastructure, or CLI) that enables an object model specification in a platform agnostic and programming language independent manner. The sheer volume reflecting the rich functionality makes it impractical to enumerate all the features of CSTA. Inspired by the success of ECMA TR/85, this TR will focus the discussion in the areas of call control and interactive voice services where the demand for CSTA object model seems to be particularly strong. Examples highly parallel to ECMA TR/85 are given in ECMA-334 (C#), a member of the CLI family.

2 References

This TR refers to CSTA and CLI, for which the following Ecma Standards are definitive references:

- ECMA-269, Services for Computer Supported Telecommunications Applications (CSTA) Phase III, 6th Edition.
- ECMA-323, XML Protocol for Computer Supported Telecommunications Applications (CSTA) Phase III, 3rd Edition.
- ECMA-348, Web Service Description Language (WSDL) for CSTA Phase III, 2nd Edition.
- ECMA TR/85, Using ECMA-323 (CSTA XML) in a Voice Browser Environment.
- ECMA-334, C# Language Specification, 2nd Edition.
- ECMA-335, Common Language Infrastructure (CLI), 2nd Edition.

The Interactive Voice Services in CSTA closely follow the operational model of Speech Application Language Tags (SALT), a specification of voice browsers available at www.saltforum.org where additional information on interoperating with CSTA can be found.

3 Brief Review of CSTA Operational Model

As illustrated in Figure 6-1 of ECMA-269, a CSTA application can draw services from Switching Function (SF) and Computing Function (CF). Typically, the CF implements the application logic, where the SF provides the infrastructure for the needed telecommunication functionality such as call controls. In addition, a CSTA application can use services from Special Resource Function (SRF) that consists of feature add-on’s to the application. Examples of features that can be implemented as SRF include Voice services and events in CSTA (ECMA-269, Clause 26). These additional features can be modelled either as part of SF or CF. In a call centre, the business procedures are often codified in and enforced by the CF software, whereas the call controls and computer telephony integration (CTI) are facilitated through service requests to the SF or SRF.

Note that, although the rest of this TR will focus on call controls and interactive voice services, they are just a category of services in CSTA SF and SRF, respectively. Examples of other categories of

SF services include capability exchange (feature discovery) services; call routing services, services to control a device (e.g. message waiting, writing to display, forwarding settings), and many others.

4 Key Concepts of Service and Object Models

Since the dawn of the computer age, software has been developed using the service model, where reusable sequence of computer instructions are packaged into procedures or subroutines, each of which offers a specific service to the rest of the program. Usually, each service operates on a series of parameters (arguments) whose roles and the transformations inflicted upon them are well defined by the service. The service model is not limited to programs running on the same computer. The service, for example, can be offered off from software in another computer connected through a network to the computer asking for the service. As a matter of fact, the majority of distributed processing protocols today are still specified in a service model.

A key feature of a service based programming model is that software designers would adopt a “vowel centric” or “action oriented” way of thinking. This is because each service is often characterized as performing a transformation on its arguments, and the transformation can usually be named after a verb while its arguments, nouns.

Grouping a sequence of operations into a packaged service is a special case of abstraction where software designers are allowed to create customized *vowels* to describe their application domains. The idea of grouping can be applied to data as well. There, relevant pieces of information can be grouped into a data structure and be thought of as an integrated entity. Similar in notion to the customized vowels, being able to define data structures allow the programmers to create customized *nouns* for their application domains. These customizations help bridging the gap between the programming model and the real world application, and therefore make the software easier to design, develop, and maintain.

As the software gets more sophisticated, it is becoming more challenging to keep a holistic view of the whole application domain. As a result, computer programs are increasingly made of well structured building blocks that themselves are computer programs composed of their own local services and data structures. Such a software design approach is reflected in the object oriented programming style. An object is an encapsulation of a data structure and the associated services and operations that can be performed on the data. The objects serve as the building blocks of a program. An object oriented programmer usually thinks in terms of what objects are involved in the application domain, and how their individual services can be collaborated to facilitate desired outcomes. Because objects often refer to things, object oriented programming has a noun centric way of thinking.

Encapsulation is a key concept to give rise to the strength and popularity of object oriented programming. Modern programming environments have further advanced the notion of encapsulation to include not only the data and operations, but also the asynchronous responses of an object as well. ECMA-334 (C#), for example, is a programming language in the ECMA-335 (CLI) family that provides comprehensive supports for object oriented programming. While data and operations can be modelled as the properties and methods of an object, respectively, C# allows the asynchronous responses to be modelled as events of an object. Localizing events to the object level streamlines the software design process by alleviating the burden of tracking event sources from the programmers.

Although CSTA has been specified largely in the form of a service model, the underlying objects are quite clear. CSTA Switching Functions are defined in terms of the interplay of Call, Connection, and the Device, and CSTA Special Resource Functions, Voice Unit, Listener, Prompt, DTMF Parser, and Prompt Queue. These abstractions lay a solid foundation for a CSTA object model because, in essence, the basic objects of CSTA have been clearly identified. The challenges, as common to all the exercises to evolve a computational platform from a service model to an object model, are to properly encapsulate the attributes, services, and events into these basic objects without departing significantly from the operational model already defined and widely implemented.

This TR strives to provide a stepping-stone in that direction. As object model design is far from a trivial task, this TR by no means is intended to be the final authority on the CSTA object model. Instead, the purpose of this TR is to summarize the collective thinking of experts in the field at the

time of publication, and to serve as an invitation to those skilled in the art to join our ongoing discussion and refine the object model design.

5 CSTA Object Model

The objects to facilitate CSTA SF have been clearly specified in Clause 6.1, ECMA-269 as the CSTA Connections, Calls, and Devices. They are accessible through a CSTA Provider that manages the collections and services surrounding these objects. In addition, the application can elicit services from Voice Resources, such as Voice Unit, Listener, Prompt, DTMF, and Prompt Queue (Clause 6.1.1.4.7 through 6.1.1.4.12, ECMA-269). Figure 1 shows the object hierarchy.

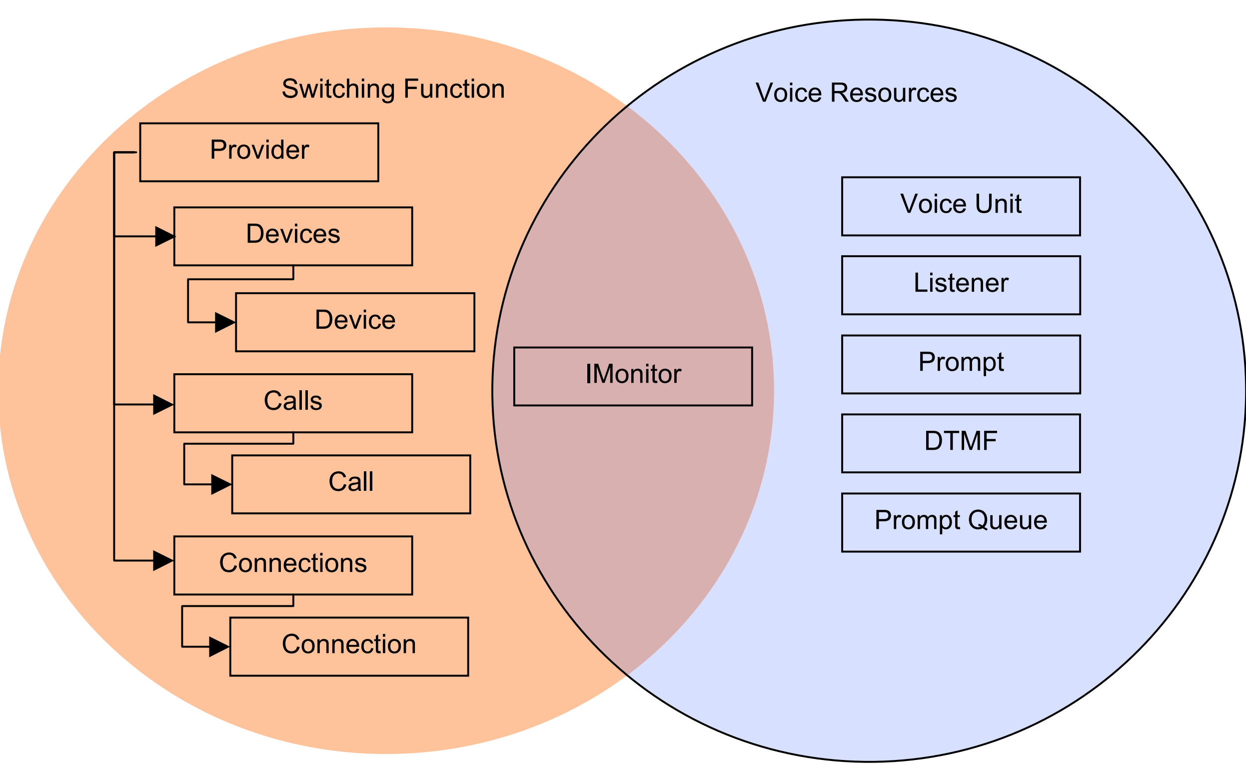

Figure 1 - Schematic of CSTA Object Model

CSTA Monitor is a special object that enables event reporting. A Monitor can be placed, for example, on Device or Call objects to observe and keep track of the states of the objects. In addition, all the SRF objects may raise events. Event monitoring is therefore a feature that objects from both SF and SRF domains will provide. Accordingly, CSTA Monitoring Services are modelled as a CLI interface that can be implemented by other objects.

5.1 CSTA Objects

Provider object models the CSTA SF subdomain. The instantiation of a Provider object establishes the application association with CSTA, i.e., the constructors of the Provider object include the implicit/explicit and client/system initiative association as defined in Clause 7 of ECMA-269. Naturally, the Provider object encapsulates the System wide Services and Events,

including Get Switching Function Capabilities, Get Switching Function Devices (13.1.4 through 13.1.6, ECMA-269), and System Services (Clause 14, ECMA-269).

The Device object models a CSTA device (6.1.1). Typically, applications do not instantiate the Device object directly. Instead, the “Get Switching Function Devices Service” operation from the Provider object returns instances of Device objects. Device objects provide operations directly related to CSTA devices, such as Get Logical/Physical Device Information (13.1.2 and 13.1.3), Snapshot Device (16.1.2), Physical and Logical Device Features (Clause 21, 22, 23), as well as device oriented call control operations like Alternate Call (17.1.2), Conference Call (17.1.9), Consultation Call (17.1.10), Group Pickup Call (17.1.14), Join Call (17.1.17), Make Call (17.1.18), Make Predictive Call (17.1.19).

The Connection object models a CSTA connection (6.1.3). Typically, applications do not instantiate the Connection objects directly as they often result from a CSTA service request invocation. For instance, when an application makes an outbound call, the resulting Connection object is accessible through an argument in the Originated Event (17.2.14). Similarly, the application can obtain the Connection object for an incoming call through the argument in the Delivered Event (17.2.5). The Connection object provides references to its Device and Call objects as it represents the relationship between these objects. It contains the connection state, and offers most of the non device oriented call control services such as Accept Call (17.1.1), Answer Call (17.1.3), Clear Connection (17.1.8), Deflect Call (17.1.11), Hold Call (17.1.15), Intrude Call (17.1.16), and Single Step Conference (17.1.24), Single Step Transfer (17.1.25), and Transfer (17.1.26).

The Call object models a CSTA call (6.1.2) that contains call attributes such as user data, correlator data, and account or charging information. Typically, applications do not instantiate the Call objects directly. Instead, a Call object is often obtained from the Connection object that models the relationship of the call and the device. Call oriented operations include the Snapshot Call (16.1.1), Snapshot CallData (16.1.3), Clear Call (17.1.7), and Services in the Call Associated Features (18.1). Note that the Call object may serve as a rich event source because CSTA allows call type monitoring. All the call-type monitoring events are raised on the Call object.

The Voice objects are very well encapsulated in ECMA-269 already. The Voice Unit object that manages voice mail and the interactive voice objects such as Listeners, Prompt, DTMF, and PromptQueue objects, have services and events described in Clause 26 in ECMA-269.

5.2 Services and Events

Services are defined as CSTA operations that can be used to observe and control objects in the CSTA SF or SRF domains. Although CSTA Services appear to have a global scope, the majority of them are indeed sponsored by a specific type of object, which often manifests itself as one of the mandatory objects in the service request parameter. As far as encapsulation is concerned, the logical choice is to associate the service with respect to its sponsoring object.

These are categories of CSTA Services (Clause 9.1, ECMA-269):

- SF Services, where SF is the server and CF is the client.
- CF Services, where SF is the client and CF is the server.
- SRF Services, where SRF is the server, and CF as the client.
- Bidirectional Services, where either the SF/SRF or the CF may be the client.

From the viewpoint of object oriented design, the services for which the CF plays the client role are conventionally modelled as the methods of the objects, whereas requests for which CF is the server are usually modelled as call back functions. Following the style of CLI, it is reasonable to use the CLI event mechanism for callbacks. In other words, both SF and SRF Services can be regarded as object methods and the CF Services, as CLI object events. A bidirectional service would be modelled with both a method and an event, adhering to the naming conventions of ECMA-335 as appropriate.

CSTA Events are exclusively directed at CF. The Monitor cross reference id often serves as a good indicator as to which Device or Call object the event should be raised and, based on service

requests, which type of event sequence will be generated. As a result, they are modelled as object events, with the event information being delivered in a System.EventArgs object. Since all the CSTA events include Event Cause, CSTASecurityData and CSTAPrivateData, a base class for all the CSTA events are defined as:

```
public CSTAEventArgs: System.EventArgs
{
     EventCause        Cause;
     CSTASecurityData  Security;
     CSTAPrivateData   PrivateData;
}
```

The XML structure for these data are defined in ECMA-323 Clause 9.7 and 9.18, and have a straightforward mapping to ECMA-335. For example, EventCause is an enumeration of CSTA event causes as follows:

```
public enum EventCause {
     ACDBusy, ACDForward, ...,
     NewCall, NextMessage, ...
     UnknownOverflow};
```

5.3 Service Acknowledgements

All CSTA Service requests will receive negative acknowledgements if parameters are invalid or the requests cannot be honoured. Clause 9.3 of ECMA-269 defined the types of errors that lead to negative acknowledgements. Similarly, the majority of service requests also receive positive acknowledgements, either in the atomic (9.2.1.1) or multistep model (9.2.1.2). While the former returns after completion, the latter requires the computing function to monitor related events to verify the request's completion.

Since the service requests are modelled as object methods, it is straightforward to treat the negative acknowledgements as exceptions that applications can catch, and the immediate positive acknowledgements as the return value of the object method call.

6 Illustrative Call Control Examples

This clause provides examples of ECMA-323 based instance documents and the corresponding object model for call control, with the same scope as ECMA TR/85, "Using ECMA-323 (CSTA-XML) in a Voice Browser Environment." Examples include:

- starting a CSTA application with implicit association
- placing a monitor on a device
- notification of an inbound alerting connection
- answering an inbound alerting connection, notification of a connected connection
- clearing a connection, notification of a cleared connection
- initiating an outbound call
- call transfer scenario
- conference scenario.

6.1 Starting a CSTA application with implicit association

Before CSTA session can be started, application association must be first established. CSTA defines several methods for application association, one of which is the implicit, CF initiative

association that is forged when the CF sends the Request System Status service request using an implicit transport mechanism to the SF. When the CF makes the request, no mandatory parameter is required. The corresponding ECMA-323 based instance document can therefore be as simple as:

```
<RequestSystemStatus
 xmlns="http://www.ecma-international.org/standards/ecma-323/csta/ed3"/>
```

Typically, the service provider will return the system status or a negative acknowledgment, upon which the application association is considered established. The system status may further indicate whether the system is disabled or enabled and ready to offer CSTA services. Clause 12.2.25 of ECMA-269 provides further descriptions of the system status values.

The same application association can be achieved in the object model by instantiating the Provider object which will supply proper constructors for possible means of associations. In this example, the sample C# code is:

```
CSTA.Provider myProvider;
try {
    myProvider = new CSTA.Provider(new System.Uri(" … URI for the SF"));
    … // myProvider.SystemStatus has the status
} catch (…) {
    // handle negative acknowledgement here
}
```

The simple constructor will send the Request System Status service request, and store the system status as a property of the Provider object for future reference.

Without the loss of generality and clarity, the following examples will omit the XML and the C# namespace declarations (xmlns and CSTA in the above example) and the exception handling for negative acknowledge for a terse presentation.

6.2 Placing a monitor on a device

In order to perform call controls, the application must obtain CSTA devices as the operating units. A CSTA device may be obtained using the Get Switching Function Devices service. Without any parameters, the service returns a list of available devices. The following example, however, demonstrates how a specific device can be requested if the device ID has been given to the application. The service request in an ECMA-323 based instance document for obtaining a device is:

```
<GetSwitchingFunctionDevices>
    <requestedDeviceID>12345</requestedDeviceID>
</GetSwitchingFunctionDevices>
```

The positive response in an ECMA-323 based instance document:

```
<GetSwitchingFunctionDeviceResponse>
    <ServiceCrossRefID>xxx</ServiceCrossRefID>
</GetSwitchingFunctionDeviceResponse>
```

returns the correlator to associate the subsequent service request. The correlator can be encapsulated into the Provider object so as to minimize the management efforts for the application developers. The corresponding C# code can therefore be:

```
myDevice = myProvider.GetDevice("12345");
```

Device type monitoring is very useful for CSTA applications. It enables the application to be notified with the current states of the monitor object through CSTA events. In order to do so, a monitor must be placed on the device by using the Monitor Start service request. The mandatory

parameter for the service request is a monitor object, and the corresponding ECMA-323 based instance document is:

```
<MonitorStart>
    <MonitorObject>
        <deviceObject>12345</deviceObject>
    </MonitorObject>
</MonitorStart>
```

The service response returns a cross reference ID for the monitor:

```
<MonitorStartResponse>
    <monitorCrossRefID>xxx</monitorCrossRefID>
</MonitorStartResponse>
```

As in the case of the service correlator, the Monitor cross reference ID can be encapsulated in the Device object where the Monitor is placed. By implementing the IMonitor interface, the Device object has the method to start the monitor, as in the following example:

```
myDevice.MonitorStart();
```

Note that starting the monitor will cause CSTA events to be reported immediately. As a result, it is advisable in the object model programming to first attach the event handlers (see following sections for examples) before starting the monitor.

6.3 Notification of an inbound alerting connection

When an inbound call arrives, the CSTA Delivered event is reported with rich information about the caller such as ANI and DNIS. Most importantly, the event also reports the connection ID created by the system to answer the call with.

The ECMA-323 based instance document of a Delivered event looks like the following:

```
<DeliveredEvent>
    <monitorCrossRefID>xxx</monitorCrossRefID>
    <connection>
        <callID>xxx</callID>
        <deviceID>12345</deviceID>
    </connection>
    <alertingDevice>
        <deviceIdentifier>12345</deviceIdentifier>
    </alertingDevice>
    <callingDevice>
        <deviceIdentifier>14085551212</deviceIdentifier>
    </callingDevice>
    <calledDevice>
        <deviceIdentifier>12345</deviceIdentifier>
    </calledDevice>
    <lastRedirectionDevice>
        <notRequired/>
    </lastRedirectionDevice>
        <localConnectionInfo>alerting</localConnectionInfo>
```

```
    <cause>newCall</cause>
    <networkCallingDevice>
          <deviceIdentifier>14085551212</deviceIdentifier>
    </networkCallingDevice>
    <networkCalledDevice>
          <deviceIdentifier>18001234567</deviceIdentifier>
    </networkCalledDevice>
    <associatedCallingDevice>
          <deviceIdentifier>023</deviceIdentifier>
    </associatedCallingDevice>
</DeliveredEvent>
```

For the object model, this event is raised at the Device object that is at the endpoint of an alerting connection for inbound call, or the device that makes the outbound call. The handler is attached as follows to receive the event:

```
myDevice.Delivered += new DeliveredEventHandler(onDelivered);
```

where myHandler, following the CLI convention, is a function that has the signature:

```
void onDelivered(object sender, DeliveredEventArgs args)
```

The information carried by this event is available in the event argument DeliveredEventArgs:

```
public class DeliveredEventArgs : CSTAEventArgs
{
    CSTA.Connection Connection;
    CSTA.Device AlertingDevice;
    CSTA.Device CalledDevice;
    CSTA.Device CallingDevice;
    CSTA.Device LastRedirectionDevice;
    CSTA.Device NetworkCallingDevice;
    CSTA.Device NetworkCalledDevice;
    CSTA.Device AssociatedCallingDevice;
    ...
}
```

In the above example, when an incoming call arrives, myHandler will be called, and the calling device and the connection, for example, can be programmatically read out from args.CallingDevice and args.Connection, respectively.

6.4 Answering an inbound alerting connection

After the application is notified of an incoming call, the Answer Call service can be used to answer the call. The mandatory parameter for the service request is the connection ID. The service request in an ECMA-323 based instance document can be:

```
<AnswerCall>
    <callToBeAnswered>
          <callID>xxx</callID>
          <deviceID>12345</deviceID>
    </callToBeAnswered>
</AnswerCall>
```

There is no mandatory parameter in the service response. Therefore, the positive acknowledgement in an ECMA-323 based instance document can be as simple as:

```
<AnswerCallResponse/>
```

For the object model, the Answer Call service is available as a method on the Connection object, which logically follows the CSTA modelling concept and encapsulates the connection ID as an inseparable pair of a device ID and a call ID. Continuing the example in 6.3, the equivalent C# code may be:

```
Connection myConnection;
...
void onDelivered(object sender, DeliveredEventArgs args)
{
    myConnection = args.Connection;
    ... // other processing on the incoming call
    if (myConnection.LocalConnectionState ==
          Connection.ConnectionState.Alerting)
          myConnection.AnswerCall( );
}
```

6.5 Clearing a connection

Once an application is done with a call, it can hang up the phone by sending Clear Connection service request.

6.5.1 Service Request

The mandatory parameter for Clear Connection is the connection ID. In an ECMA-323 based instance document, this appears as:

```
<ClearConnection>
   <connectionToBeCleared>
         <callID>xxx</callID>
         <deviceID>12345</deviceID>
   </connectionToBeCleared>
</ClearConnection>
```

The service response has no mandatory parameter. The ECMA-323 based instance document therefore can be as simple as:

```
<ClearConnectionResponse/>
```

The service is modelled as a method on the Connection object. The above ECMA-323 snippets are equivalent to:

```
myConnection.ClearConnection();
```

6.5.2 Notification of a cleared connection

When the telephony platform finishes clearing the connection, it sends the CSTA Connection Cleared event. The event is reported through the monitors placed on the call, or on the devices involved in the call. For either case, the ECMA-323 based instance document is:

```
<ConnectionClearedEvent>
   <monitorCrossRefID>99</monitorCrossRefID>
   <droppedConnection>
         <callID>1</callID>
         <deviceID>12345</deviceID>
   </droppedConnection>
   <releasingDevice>
         <deviceIdentifier>12345</deviceIdentifier>
   </releasingDevice>
   <cause>normalClearing</cause>
</ConnectionClearedEvent>
```

Following the previous example for the object model, the event will be reported on the releasing device if an event handler has been attached:

```
myDevice.ConnectionCleared +=
new ConnectionClearedEventHandler(myHandler1);
...
void myHandler1(sender object, ConnectionClearedEventArgs args)
{ ... }
```

The ConnectionClearedEventArgs consists of the following information:

```
public class ConnectionClearedEventArgs : CSTAEventArgs
{
   Connection DroppedConnection;
   Device     ReleasingDevice;
   ...
}
```

In this example, args.ReleasingDevice will be the same as myDevice, and args.Cause will be equal to CSTA.EventCause.NormalClearing.

Note that the Connection Cleared event will also be raised if the far end hangs up the phone, for which case the args.ReleasingDevice will not be the same as myDevice.

6.6 Initiating an outbound call

The Make Call service can be used to initiate an outbound call. The mandatory parameters are the device IDs of the call initiating and the targeted recipient devices. Note that by default, the connection to the calling device is not automatically answered unless an optional parameter autoOriginate is set to “do not prompt.”

6.6.1 Service Request

The Make Call service with do not prompt option can be issued as:

```
<MakeCall>
   <callingDevice>12345</callingDevice>
   <calledDirectoryNumber>18005551212</calledDirectoryNumber>
   <autoOriginate>doNotPrompt</autoOriginate>
</MakeCall>
```

The service response returns the initial connection created by the service:

```
<MakeCallResponse>
   <callingDevice>
         <callID>xxx</callID>
         <deviceID>12345</deviceID>
   </callingDevice>
</MakeCallResponse>
```

The corresponding service is available as the object method on the calling device:

```
myInitialConnection =
myDevice.MakeCall("18005551212", AutoOriginate.DoNotPrompt);
```

where AutoOriginate is an enumeration as:

```
public enum AutoOriginate {Prompt, DoNotPrompt};
```

6.6.2 Outbound Call Event Sequence

An outbound call usually involves a series of events that report the progress of the call, including Originated, Network Reached, Delivered and Established events. Applications may choose only to handle some but not all of these events.

6.6.2.1 Originated event

The event indicates the call is being made, which is useful especially when the default Make Call behaviour for the calling device is prompt to connect. The mandatory parameters for this event are the called and calling device IDs, the monitor cross reference ID, the originated connection ID, and the event cause, which in an ECMA-323 based instance document appears as:

```
<OriginatedEvent>
 <monitorCrossRefID>99</monitorCrossRefID>
 <originatedConnection>
       <callID>xxx</callID>
       <deviceID>12345</deviceID>
 </originatedConnection>
 <callingDevice>
       <deviceIdentifier>12345</deviceIdentifier>
 </callingDevice>
 <calledDevice>
       <deviceIdentifier>18005551212</deviceIdentifier>
 </calledDevice>
 <cause>makeCall</cause>
```

```
</OriginatedEvent>
```

In C#, the corresponding structure can be modelled as:

```
public OriginatedEventArgs: CSTAEventArgs
{
 Connection OriginatedConnection;
 Device      CallingDevice;
 Device      CalledDevice;
 ...
}
```

6.6.2.2 Networked Reached event

When the target recipient is outside the CSTA domain, the Networked Reached event will be raised when the call reaches the network interface. In addition to the common monitor cross reference ID, the caller's and recipient's device IDs, the event will also report the connection ID for the connection associated with the network interface, and the ID for the last redirection device.

An ECMA-323 based instance document is as follows:

```
<NetworkReachedEvent>
 <monitorCrossRefID>99</monitorCrossRefID>
 <outboundConnection>
       <callID>2</callID>
       <deviceID>023</deviceID>
 </outboundConnection>
 <networkInterfaceUsed>
       <deviceIdentifier>023</deviceIdentifier>
 </networkInterfaceUsed>
 <callingDevice>
       <deviceIdentifier>12345</deviceIdentifier>
 </callingDevice>
 <calledDevice>
       <deviceIdentifier>18005551212</deviceIdentifier>
 </calledDevice>
 <lastRedirectionDevice>
       <notRequired/>
 </lastRedirectionDevice>
 <cause>normal</cause>
</NetworkReachedEvent>
```

The corresponding C# structure is

```
public NetworkReachedEventArgs: CSTAEventArgs
{
 Connection OutboundConnection;
 Device      NetworkInterfaceUsed;
 Device      CallingDevice;
```

```
    Device      CalledDevice;
    Device      LastRedirectedDevice;
    ...
   }
```

6.6.2.3 Delivered event

The Delivered event is similar to the example shown in 6.3, except here it means a connection has been made with the recipient and the connection is in alerting. The event cause for outbound call is usually EventCause.NetworkSignalling.

6.6.2.4 Established event

The Established event is raised when the recipient picks up the phone. Raising the event is not always possible when the recipient is on a network other than the caller’s.

The structure for the Established event is very similar to Delivered event, and therefore an example is not shown here.

6.7 Single step transfer or Deflect call

The CSTA Single Step Transfer service transfers a call from a device to another device, disconnecting existing connection in the process. The Deflect Call service performs the similar task, one most notable difference being that Deflect Call service retains the same Call ID after transfer, while a new one will be assigned for the Single Step Transfer.

6.7.1 Service Requests

The mandatory parameters for the service are the connection to be transferred, and the device ID for the transfer target. An ECMA-323 based instance document is as follows:

```
<SingleStepTransferCall>
  <activeCall>
        <callID>xxx</callID>
        <deviceID>12345</deviceID>
  </activeCall>
  <transferredTo>333333</transferredTo>
</SingleStepTransferCall>
```

The service response returns the connection to the device where the call is transferred. An ECMA-323 based instance document is as follows:

```
<SingleStepTransferCallResponse>
  <transferredCall>
        <callID>yyy</callID>
        <deviceID>333333</deviceID>
  </transferredCall>
</SingleStepTransferCallResponse>
```

Similarly, an ECMA-323 based instance document for Deflect Call service may be:

```
<DeflectCall>
  <callToBeDiverted>
        <callID>1</callID>
        <deviceID>22343</deviceID>
  </callToBeDiverted>
```

```
   <newDestination>333333</newDestination>
</DeflectCall>
```

Because there is no mandatory parameter in the response, the ECMA-323 based instance document for Deflect service response can be as simple as:

```
<DeflectCallResponse/>
```

The equivalent service for Single Step Transfer is modelled as a method on the Connection object. As a result, a C# sample code may appear as:

```
newDestination = myProvider.GetDevice("333333");
newConnection= myConnection.SingleStepTransfer(newDestination);
```

Because Deflect Call is more a connection oriented operation, the service is modelled as a method on the Connection object. A C# sample is as follows:

```
myConnection.Deflect(newDestination);
```

6.7.2 Notification events

In addition to the service response for Single Step Transfer where the application can obtain the new connection for the transferred call, CSTA also raises event on the device to signal its call has been transferred and connection severed. An ECMA-323 based instance document for the Transferred event is:

```
<TransferedEvent>
   <monitorCrossRefID>99</monitorCrossRefID>
   <primaryOldCall>
         <callID>xxx</callID>
         <deviceID>12345</deviceID>
   </primaryOldCall>
   <transferringDevice>
         <deviceIdentifier>12345</deviceIdentifier>
   </transferringDevice>
   <transferredToDevice>
         <deviceIdentifier>333333</deviceIdentifier>
   </transferredToDevice>
   <transferredConnections>
         <connectionListItem>
               <oldConnection>
                     <callID>xxx</callID>
                     <deviceID>12345</deviceID>
               </oldConnection>
         </connectionListItem>
   </transferredConnections>
   <cause>singleStepTransfer</cause>
</TransferedEvent>
```

The equivalent structure in C# can be modelled as:

```
public TransferredEventArgs: CSTAEventArgs
{
   Connection  PrimaryOldCall;
   Device      TransferringDevice;
   Device      TransferredToDevice;
   ConnectionList TransferredConnections;
}
```

where the ConnectionList is a property that implements System.Collections.IList interface. If the call is moved using the Deflect Call service, a Diverted event will be raised. An ECMA-323 based instance document is:

```
<DivertedEvent xmlns="http://www.ecma.ch/standards/ecma-323/csta/ed2">
   <monitorCrossRefID>99</monitorCrossRefID>
   <connection>
         <callID>xxx</callID>
         <deviceID>12345</deviceID>
   </connection>
   <divertingDevice>
         <deviceIdentifier>12345</deviceIdentifier>
   </divertingDevice>
   <newDestination>
         <deviceIdentifier>333333</deviceIdentifier>
   </newDestination>
   <cause>redirected</cause>
</DivertedEvent>
```

and the corresponding C# event argument is

```
public DivertedEventArgs : CSTAEventArgs
{
   CSTA.Connection   Connection;
   CSTA.Device       DivertingDevice;
   CSTA.Device       NewDestination;
   ...
}
```

6.8 Single step conference or Join Call

It is quite common that another party must be added to the call. This operation may be achieved through either Single Step Conference or Join Call service in CSTA. Although the outcome may appear similar, Single Step Conference is conceptually more like adding a device to an existing connection, whereas Join Call, adding a connection to a device.

Both services allow the app to specify the participation type of the added device. The possible types include active and silent: Active participation, which is default, allows the new device to receive and send audio to the call, while silent participation allows only receiving audio from the call.

Because the Join Call service functions like making a new connection from a device to a call, the service allows the application to specify AutoOriginate type as in the case for Make Call (see 6.6).

6.8.1 Service Requests

The mandatory parameters for Single Step Conference are the IDs for the added device and connection. An ECMA-323 based instance document is:

```
<SingleStepConferenceCall>
   <activeCall>
        <callID>1</callID>
          <deviceID>22343</deviceID>
   </activeCall>
   <deviceToJoin>55555</deviceToJoin>
</SingleStepConferenceCall>
```

The service response returns the new connection being created for the new device:

```
<SingleStepConferenceCallResponse>
   <conferencedCall>
          <callID>1</callID>
          <deviceID>55555</deviceID>
   </conferencedCall>
</SingleStepConferenceCallResponse>
```

The service is modelled as a method on the Connection object for which the new device will be added. It returns the new connection established for the new device, as in the C# sample below:

```
newDevice = myProvider.GetDevice("555555");
newConnection = myConnection.SingleStepConference(newDevice);
```

The mandatory parameters for Join Call are the same, as the service response. The ECMA-323 based instance documents are therefore omitted here. However, this service is modelled as a method on the Device object to be added to the call. The corresponding C# code is therefore somewhat different in that one has to obtain the object representing the added device first:

```
newConnection = newDevice.Join(myConnection);
```

6.8.2 Notification Event

As a result of Single Step Conference or Join Call, the Conferenced event may be raised to indicate a new device has been added to the call, after which other events such as Established event may follow when the new device answers the call.

An ECMA-323 based instance document for Conferenced event is as follows:

```
<ConferencedEvent>
   <monitorCrossRefID>99</monitorCrossRefID>
   <primaryOldCall>
          <callID>1</callID>
          <deviceID>22343</deviceID>
   </primaryOldCall>
   <conferencingDevice>
          <deviceIdentifier>22343</deviceIdentifier>
   </conferencingDevice>
```

```
    <addedParty>
          <deviceIdentifier>55555</deviceIdentifier>
    </addedParty>
    <conferenceConnections>
          <connectionListItem>
<newConnection>
                        <callID>1</callID>
                        <deviceID>22343</deviceID>
</newConnection>
<endpoint>
                      <deviceID>22343</deviceID>
                <endpoint>
          </connectionListItem>
          <connectionListItem>
                <newConnection>
                        <callID>1</callID>
                        <deviceID>33333</deviceID>
                </newConnection>
                <endpoint>
                      <deviceID>33333</deviceID>
                </endpoint>
          </connectionListItem>
          <connectionListItem>
                <newConnection>
                        <callID>1</callID>
                        <deviceID>55555</deviceID>
                </newConnection>
                <endpoint>
                      <deviceID>55555</deviceID>
                </endpoint>
          </connectionListItem>
    </conferenceConnections>
    <cause>singleStepConference</cause>
</ConferencedEvent>
```

The corresponding structure in C# is:

```
public ConferencedEventArgs : CSTAEventArgs
{
   Connection        PrimaryOldCall;
   Device            ConferencingDevice;
   Device            AddedParty;
   ConnectionList    Connections;
   ...
}
```

7 Illustrative Examples for Interactive Voice Functions

This section provides ECMA-323 based instance document and the corresponding C# examples for interactive voice functions. The examples include:

- obtaining an object
- simple prompt playback
- advanced prompt playback using subqueue
- prepare speech recognition
- concurrent DTMF and speech recognition
- concurrent speaker verification alongside with speech recognition

7.1 Obtaining an object

A Prompt object can be instantiated in C# as:

```
Prompt myPrompt = new Prompt( );
```

or obtained through Get Switching Function Devices service. The following ECMA-323 based instance document uses this service to obtain a collection of devices that can provide the Prompt functionality:

```
<GetSwitchingFunctionDevices>
    <requestedDeviceCategory>Prompt</requestedDeviceCategory>
</GetSwitchingFunctionDevices>
```

The corresponding C# sample would appear like the following:

```
System.Collection prompts = myProvider.GetDevices(DeviceCategory.Prompt);
```

where

```
public enum DeviceCategory = { ACD, GroupACD, GroupHunt, … ,
    GenericVoice, VoiceUnit, Listener, Prompt, DTMF, PromptQueue};
```

7.2 Attaching the audio

The voice devices interact with the callers via voice and therefore must be associated with a CSTA Connection before their services can be utilized. The CSTA Join Call or Single Step Conference Call services demonstrated in 6.8 can be used to indicate the connection for the interactive voice devices to send or draw audio from. For example, a Prompt object can be attached with an audio channel for playback using either

```
promptConnection = myConnection.SingleStepConference(myPrompt);
```

or

```
myPrompt.Join(myConnecction);
```

Note that it is typical that each interactive voice object is narrowly constructed to handle a single audio channel, i.e., one Connection. If this is the case, the above step can be combined with object instantiation in 7.1, as the following example demonstrates:

```
myPrompt = new Prompt(myConnection);
```

It is, however, possible to share a device with multiple connections because the majority of voice services and events are connection oriented. This advanced scenario is further discussed in 7.8.

7.3 Simple prompt playback

The Start service can be used to play a text to speech synthesized voice. The mandatory parameter for the service is the connection to which the voice will be played. Optionally, the service accepts a string that represents the text to be synthesized.

The ECMA-323 based instance document for saying the sentence “Thank you for calling!” is as follows:

```
<Start>
    <overConnection>
        <callID>xxx</calledID>
        <deviceID>12345</deviceID>
    </overConnection>
    <text>Thank you for calling!</text>
</Start>
```

There is no mandatory parameter for the service response. As a result, the response can be as simple as:

```
<StartResponse/>
```

The service can be modelled as a method on the voice objects. As a result, a C# example is:

```
myPrompt.Start(“Thank you for calling!”);
```

The application may place a monitor on the device to receive the CSTA Completed or Bookmark Reached event. This can be achieved following the same approach in 6.2. As is in the previous case, the mandatory monitor cross reference ID and connection ID can be entirely encapsulated in the object model. Accordingly, the CSTAEventArgs can be used for Completed event where the Bookmark Reached event contains the additional bookmark:

```
public BookmarkReachedEventArgs : CSTAEventArgs
{
    string            Bookmark;
}
```

7.4 Advanced prompt playback

It is very common that an application would like to queue up multiple prompts and play them together. Prompt Queue and the corresponding Queue service may be used for this kind of scenarios. A Prompt Queue device can be obtained and attached to the call in the same manner as shown in 7.1 and 7.2. In the following, we demonstrate a scenario where a filler prompt is continuously played to the caller when the application retrieves relevant data, and is only stopped while the app is ready to present the data to the caller.

7.4.1 Set Voice Attribute for the filler prompt

In this example, the filler prompt is a combination of synthesized text and music. This can be achieved by setting the innerXML attribute of a Prompt using the Set Voice Attribute service. An ECMA-323 based instance document is:

```
<SetVoiceAttribute>
  <connection>
        <callID>xxx</callID>
        <deviceID>12345</deviceID>
  </connection>
  <innerXML>
        <speak xmlns="http://www.w3.org/2001/10/synthesis">
              Please stay on the line while I retrieve the data.
              <audio src="urn:my audio source"/>
        </speak>
  </innerXML>
</SetVoiceAttribute>
```

There is no mandatory parameter for the positive acknowledgement. As a result, the service response can simply be:

```
<SetVoiceAttributeResponse/>
```

Voice Attributes are modelled as object properties. Setting and reading the properties are equivalent to issue the Set Voice Attribute and Query Voice Attribute service requests, respectively. As a result, the corresponding C# example is:

```
fillerPrompt.InnerXML = "<speak ...> ... </speak>";
```

The content of the innerXML attribute is a string representation of W3C SSML. At this time, there is no native SSML tag to refer to external contents. Such a mechanism, however, is available in the SALT specification through the use of the <content> tag. In addition to the inline SSML in the above example, application may also specify the playback content in reference. For example, using ECMA-323, one may specify the following instance document:

```
<SetVoiceAttribute>
  <connection>
        <callID>xxx</callID>
        <deviceID>12345</deviceID>
  </connection>
  <innerXML>
        <content xmlns="http://www.saltforum.org/..."
              src="realdocument.ssml"/>
  </innerXML>
</SetVoiceAttribute>
```

Similarly in C#:

```
fillerPrompt.innerXML = "<content xmlns='…' src='…'/>";
```

7.4.2 Play the filler prompt

The audio in the above example can be connected to a broadcast source where music is continuously played. Such an arrangement will lead to a prompt that does not end until explicitly stopped, using the method demonstrating in 7.4.4 below for example. Another common use

case, though, is to connect the audio to a fixed length pre-recorded waveform so that the application can decide how many filler instances should be played while the data is being prepared. Each instance of playback can be sent to the Prompt Queue using the Queue service. Using ECMA-323, the following instance document, sends an instance to the Prompt Queue:

```
<Queue>
  <overConnection>
        <callID>xxx</callID>
        <deviceID>12345</deviceID>
  </overConnection>
</Queue>
```

There is no mandatory parameter for the positive acknowledgement. The response can be as simple as:

```
<QueueResponse/>
```

When multiple instances are needed, the above ECMA-323 based instance document can be repeated as many times as appropriate. The corresponding example in C# for two instances is:

```
fillerPrompt.Queue();
fillerPrompt.Queue();
```

After queue is prepared, the playback may be initiated with the Start service shown in 7.3, using the connection ID for either the Prompt Queue or the filler prompt as the parameter. In C#, an example is as follows:

```
myPromptQueue.Start();
```

7.4.3 Prepare the new prompt

After the application fetched the data, a new prompt can be created to present the data to the caller. The application can follow the examples in 7.1 and 7.2 to set up the prompt, using examples in 7.4.1 for the content if necessary. Finally, the Start service can be used to place the new prompt to the Prompt Queue to mark the new prompt as ready. All these steps can take place while the filler prompt is still playing.

7.4.4 Stop the filler prompt

As the new prompt is ready for playback, the Stop service can be used to remove the unfinished portion of the filler prompt. The mandatory parameter is the connection ID for the Prompt Queue, as shown in this ECMA-323 based instance document:

```
<Stop>
  <connection>
        <callID>xxx</callID>
        <deviceID>23456</deviceID>
  </connection>
</Stop>
```

Since there is no mandatory parameter, the service response can be:

```
<StopResponse/>
```

The equivalent C# example is:

```
myPromptQueue.Stop( );
```

7.5 Simple speech recognition

To utilize the voice services for speech input processing such as speech recognition, the application has to first obtain the device and connection, setting the grammars for recognition, and

attaching event handlers to receive notifications. Unlike the prompt playback, the last step is required for speech input services.

7.5.1 Obtain device and attach audio

The method to obtain a Listener device and attach a connection to, is identical to Prompt case where applications may use the CSTA services as demonstrated in 7.1 and 7.2. Equivalent C# code which models Listener as add-on to CF may look like:

```
myListener = new Listener (myConnection);
```

7.5.2 Set up recognition grammars

Similar to setting the text for prompt playback, the recognition grammars are set using the Set Voice Attribute service in CSTA shown in 7.4.1. This is an ECMA-323 based instance document example:

```
<SetVoiceAttribute>
  <connection>
      <callID>xxx</callID>
      <deviceID>23456</deviceID>
  </connection>
  <grammars>
      <grammar name=”general”>
          <grammar xmlns=”http://www.w3.org/2001/06/grammar” ...>
              <rule name=”start”>
                  <one-of>
                      <item>help</item>
                      <item>operator</item>
                  </one-of>
              </rule>
          </grammar>
      </grammar>
      <grammar name=”main” src=”http://mygrammar...”/>
  </grammars>
</SetVoiceAttribute>
```

Note that, as shown in the above example, the individual grammar in the Grammars collection can be either a W3C SRGS inline grammar or a URI reference. As a result, the Grammars collection is modelled as an object property of the type System.Collections.IDictionary in ECMA-335. Two additional objects, SrgsGrammar and UriGrammar, are also provided to model each type of the grammars. Both implement a common interface IGrammarProvider that serves as the value type for the Grammars collection. The corresponding C# code equivalent to the ECMA-323 based instance document above is:

```
myListner.Grammars.Add(”general”, new SrgsGrammar(”<grammar
xmlns...”));

myListner.Grammars.Add(”main”, new UriGrammar(”http://mygrammar...”));
```

7.5.3 Attach event handlers and start recognition

Events are the mechanism for the application to obtain recognition results. A monitor therefore must be placed on the device before the CSTA Start service is issued to kick off the recognition. The ECMA-323 based instance documents for placing a monitor and issuing the Start request are shown in 6.2 and 7.3.

When a sentence is recognized, the Recognized event is raised. An ECMA-323 based instance document that recognizes the caller's sentence "check email from John Smith" can be as follows:

```
<RecognizedEvent>
   <monitorCrossRefID>123</monitorCrossRefID>
   <overConnection>
         <callID>xxx</callID>
         <deviceID>23456</deviceID>
   </overConnection>
   <result>
         <SML confidence="0.7" text="check email from john smith">
               <checkEmail confidence="..." text=...>
                     <sender confidence="0.8" text="john smith">
                           <name>John Smith</name>
                           <email>johns</email>
                     </sender>
               </checkEmail>
         </SML>
   </result>
   <text>check email from john smith</text>
</RecognizedEvent>
```

The equivalent information is reported back to the application in the following object:

```
public class RecognizedEventArgs: CSTAEventArgs
{
   string      Result;
   string      Text;
}
```

The result string can be converted into a System.Xml.XmlDocument, using the LoadXml method in ECMA-335, for example.

To receive events in the object model, the application has to attach the event handlers and activate the monitor as shown in 6.2:

```
myListener.Recognized += new RecognizedEventHandler(myHandler);
...
myListener.MonitorStart();
```

A sample code to obtain the sender's email address, for example, can be as follows:

```
void myHandler(object sender, RecognizedEventArgs e)
{
   System.Xml.XmlDocument res = new System.Xml.XmlDocument(e.Result);
   string target = res.SelectSingleNode("/*/email").InnerText;
   ...
}
```

7.6 Simultaneous DTMF and speech recognition

Callers using telephone have two modalities to interact with automatic agents: spoken language and DTMF touch tones. While the former offers a potentially richer expressive power and more natural user experience, the latter proves more robust against background noise and user variations.

In addition to the speech modality using the Listener demonstrated above, CSTA also provides touch tone parsing services via DTMF. Using DTMF follows largely the same process as speech: allocating a DTMF device with the corresponding audio channel, attaching the grammar collection and event handlers, and finally using the CSTA Start service to start the processing. The example is almost identical to 7.5 and is therefore omitted here.

The DTMF and speech modalities can be used at the same time that allows the caller to use either speech or DTMF in responding to the automatic agent. The process to set this up is straightforward: a Listener and a DTMF device will be joined to the same call, and two Start service requests are issued in tandem, one using the connection ID for the Listener and the other, for the DTMF. A C# sample is as follows:

```
myListener = new Listener(myConnection);
myListener.Grammars.Add(“main”, new UriGrammar(”...”));
myListener.RecognizedEvent += new RecognizedEventHandler(myHandler);
myListener.NotRecognizedEvent +=
    new NotRecognizedEventHandler(myNoHandler);
myDtmf = new Dtmf(myConnection);
myDtmf.Grammars.Add(“main”, new UriGrammar(“...”));
myDtmf.RecognizedEvent += new RecognizedEventHandler(myHandler);
myDtmf.NotRecognizedEvent +=
    new NotRecognizedEventHandler(myNoHandler);
...
myListener.Start();
myDtmf.Start();
...
```

CSTA requires the Listener and DTMF devices to coordinate their processings. For instance, when it becomes clear that the caller has chosen one input modality over the other, the unchosen device will automatically cease its operation (see 6.2 in ECMA-269).

7.7 Simultaneous speech recognition and speaker verification

In addition to activating multiple input modalities, CSTA allows multiple functions of the speech input modality. For example, it is possible to overlay speaker verification on top of a speech recognition session where the same audio collected from the caller is used for both purposes. The steps needed are similar to those demonstrated in 7.6, with the exception that only one Start service is needed for the devices of the same modality.

It is out of scope how the multiple functions of the speech modality are made available. Typically, the application may obtain the resource to perform a specific function either by using the CSTA capability exchange services (Clause 13, ECMA-269), or through prearranged provisioning of device IDs. An additional attribute, called server, for the Listener device is introduced so that a URI for the function provided by the device can be specified. In C# example,

```
myRecognizer = new Listener();
myRecognizer.Server = new System.Uri(“sip:mySRServer”);
myRecognizer.Grammars.Add(“recognition”, new UriGrammar(“...”));
// setting up other attributes of the speech recognizer
```

```
// ...
myVerifier = new Listener();
myVerifier.Server = new System.Uri("sip:myVerifier");
myVerifier.Grammars.Add("cohort", new UriGrammar("..."));
// setting up other attributes of the speaker verifier
// ...
myRecognizer.Join(myConnection);
myVerifier.Join(myConnection);
myRecognizer.Start();
```

As in the case of 7.6, the speech processings among the Listener devices joined to the same call are coordinated. The detail behaviours are specified in 6.2 of ECMA-269.

7.8 Sharing voice devices

It is possible that some interactive voice functions can be very resource demanding. Recognizing proper names from a large database, for example, usually requires a speech recognizer to load up a sizeable grammar. It is therefore beneficial to share such a device among multiple connections as appropriate.

The majority of voice services and events in ECMA-269 are connection oriented. Device sharing is therefore straightforward: Interactive Voice services take a connection ID as the mandatory parameter that defines which audio channel the service should be interacting with. Similarly, all Interactive Voice events report the connection ID that identifies which audio channel generates the event. ECMA-323 based instance document examples for device sharing would be identical to all the previous examples and are omitted here.

For object modelling where events and properties are encapsulated for the application developers for usability, extra cautions must be taken for such an advanced scenario. As mentioned in 7.2, for the most common usages where a voice object is usually constructed to handle one single audio channel, it is logical to encapsulate the primary connection the object is dealing with. Methods modelling the voice services can therefore require no extra arguments, as in the Start method shown above. This is not the case for sharing because the connection must be explicitly specified in the service request.

To be specific, the following sample shows how a device can be shared by two connections, using either Single Step Conference or Join Call service:

```
myIVConnection1 = myConnection1.SingleStepConference(myRecognizer);
...
myIVConnection2 = myRecognizer.Join(myConnection2);
...
```

To invoke the Start service in this case, the connection object must be explicitly specified:

```
myRecognizer.Start(myIVConnection1);
...
myRecognizer.Start(myIVConnection2);
myRecognizer.Clear(myIVConnection1);
...
```

Similarly, to handle events under the device type monitoring, the programmer can usually ignore the sender argument when the object is constructed for a singleconnection. In contrast, for the sharing scenario, the connection that causes the event must be explicitly inspected:

```
void myHandler(object sender, RecognizedEventArgs args)
{
```

```
        try {
              Connection target = (Connection) sender;
              if (target == myIVConnection1) {
                    // perform tasks for connection 1
              } else if (target == myIVConnection2) {
                    // perform tasks for connection 2
              }
        } catch (...) {
              ...
        }
}
```

Annex A
(informative)

A comparison between CSTA and SALT 1.0

The ECMA-269 Edition 6 enhancements to CSTA voice services are derived from SALT 1.0. While adapting SALT 1.0 for CSTA, many identifier name changes are needed for better compliance with the naming guidelines of ECMA-335. In this Annex, two simple C# examples, one for speech recognition and the other for synthesis, are shown to further demonstrate the transition path from SALT 1.0 to CSTA. Although SALT can be embedded into any XML document to provide speech services, the examples below focus on the environment of HTML with ECMAScript where SALT is commonly used.

A.1 Speech input

In SALT 1.0, a speech recognizer is obtained using the <listen> tag:

```
<listen id="myRecognizer"
          mode="single"
          initialtimeout="500"
          onreco="handler1()"
          onnoreco="handler2()"
          onsilence="handler3()">
    <param name="server">sip:myserver</param>
    <grammar name="global" src="http://mygrammarsrc/... "/>
</listen>
```

The corresponding C# example is:

```
myRecognizer = new Listener();
myRecognizer.Mode = Listener.Mode.Single;
myRecognizer.SilenceTimeout = 500;
myRecognizer.Recognized +=
    new RecognizedEventHandler(handler1);
myRecognizer.NotRecognized +=
    new NotRecognizedEventHandler(handler2);
myRecognizer.SilenceTimeoutExpired +=
    new SilenceTimeoutExpiredEventHandler(handler3);
myRecognizer.Server = new System.Uri("sip:myserver");
myRecognizer.Grammars.Add("global",
    new UriGrammar("http://mygrammarsrc/..."));
// need to add CSTA connection to myRecognizer
```

In both cases, the recognition is kicked off using:

```
myRecognizer.Start();
```

where C# and ECMAScript happen to have the same expression. As commented, the object model code needs to add a CSTA connection as an audio source for the recognizer. In SALT 1.0, the browser automatically does this.

Both the SALT and CSTA object model use events to receive recognition outcome. As shown in the above examples, handlers must be attached before recognition starts. Note that SALT is designed to be environment agnostic. The event handling follows the existing mechanism of SALT's hosting environment. In HTML Document Object Model (DOM) where SALT is commonly hosted, event handlers are invoked without local arguments. Instead, the relevant information is kept in the window.event object, including the source element that raises the event. Application developers can inspect the contents of the object inside the respective event handler. For the last example, the handler for the recognition event can be authored in ECMAScript as follows:

```
function handler1 ()
{
    var res = window.event.srcElement.recoresult; // SML document
    var transcript = window.event.srcElement.text; // transcription
    var temp = res.selectSingleNode("//name[@confidence > 0.5]");
          // use Xpath to retrieve the first node called 'name' with
          // desired recognition confidence score
          ...
}
```

Note that ECMAScript is not a strongly typed language. Nevertheless, it should be understood that in the above example, the recognition result recoresult, an SML document, is defined as an XML DOM Node type by SALT specification so that the programming language independent DOM API, such as the select single node method demonstrated above, may be applied to query the data. Under CLI, however, the relevant information for each event is often passed on as an argument to the event handler. As a result, the developer can directly access this information without consulting any global variable, as shown below:

```
void handler1 (object sender, RecognizedEventArgs args)
{
    XmlNode     res = args.Result;
    string      transcript = args.Text;
    XmlNode     temp = res.SelectSingleNode("//name[@confidence > 0.5]");
    ...
}
```

A.2 Speech output

In SALT 1.0, a prompt is declared using the <prompt> tag:

```
<prompt id="myPrompt"
          bargein="true"
          onbargein="handler1()"
          onbookmark="handler2()"
          oncomplete="handler3()">
    <cotent src="ding.wav"/> <mark name="first"/>
    Thank you for calling! <mark name="second"/>
    How may I help you?
</prompt>
```

The corresponding C# example is:

```
myPrompt = new Prompt();
myPrompt.AutoInterruptible = true;
myPrompt.BargeinDetected +=
    new BargeinDetectedEventHandler(handler1);
myPrompt.BookmarkReached +=
    new BookmarkReachedEventHandler(handler2);
myPrompt.Completed += new CompletedEventHandler(handler3);
myPrompt.InnerXml = "<content src="ding.wav"/> <mark name="first"/>" +
    "Thank you for calling! <mark name="second"/>" +
    "How may I help you?";
// need to add a connection
```

In either case, the prompt can be sent to Prompt Queue using

```
myPrompt.Queue();
```

or immediately started using

```
myPrompt.Start();
```

where C# and ECMAScript again have the same expression.