Chapter 1
Introduction to the Multimodal Architecture Specification

Jim Barnett

Abstract The W3C's Multimodal Architecture standard is a high-level design featuring loosely coupled components. Its goal is to encourage interoperability and re-use of components, without enforcing any particular approach to building multimodal applications. This paper offers an overview of the architecture, outlining its components and the events they use to communicate, as well as giving basic examples of how it can be applied in practice.

1.1 Overview

Many standards emerge in areas where the technology is stable and industry participants think that they understand the field well enough to be able to codify existing best practices. However the consensus within the Multimodal Interaction Working Group of the W3C was that best practices for multimodal application development had not yet emerged. The group therefore took it as its task to support exploration, rather than trying to codify any particular approach to multimodal applications. The goal of the Multimodal Architecture and Interfaces standard [1] is to encourage re-use and interoperability while being flexible enough to allow a wide variety of approaches to application development. The Working Group's hope is that this architecture will make it easier for application developers to assemble existing components to get a base multimodal system, thus freeing them up to concentrate on building their applications.

As part of the discussions that lead to the Multimodal Architecture, the group considered existing multimodal languages, in particular SALT [2] and HTML5 [3]. SALT was specifically designed as a multimodal language, and consisted of speech tags that could be inserted into HTML or similar languages. HTML5 in turn has multimodal capabilities, such as video, which were absent from earlier versions of

J. Barnett (✉)
Department of Architecture Team, Genesys, Daly City, CA, USA
e-mail: jim.barnett@genesys.com

© Springer International Publishing Switzerland 2017

D.A. Dahl (ed.), *Multimodal Interaction with W3C Standards*,
DOI 10.1007/978-3-319-42816-1_1

HTML. One problem with this approach is that it is both language- and modality-specific. For example, neither SALT nor HTML5 supports haptic sensors, nor do they provide an extension point that would allow them to be integrated in a straightforward manner. Furthermore, in both cases overall control and coordination of the modalities is provided by HTML, which was not designed as a control language. Multimodal application developers using HTML5 are thus locked into a specific graphical language with limited control capabilities and no easy way to add new modalities. As a result of these limitations, HTML5 is not a good framework for multimodal experimentation.

The Multimodal Working Group's conclusion is that it was too early to commit to any modality-specific language. For example, VoiceXML [4] has been highly successful as language for speech applications, particularly over the phone. However there is no guarantee that it will turn out to be the best language for speech-enabled multimodal applications. Therefore the Working Group decided to define a framework which would support a variety of languages, both for individual modalities and for overall coordination and control. The framework should rely on simple, high-level interfaces that would make it easy to incorporate existing languages such as VoiceXML as well as new languages that haven't been defined yet. The Working Group's goal was to make as few technology commitments as possible, while still allowing the development of sophisticated applications from a wide variety of re-usable components. Of necessity the result of the Group's work is a high-level framework rather than the description of a specific system, but the goal of the abstraction is to let application developers decide how the details should be filled in.

We will first look at the components of the high-level architecture and then at the events that pass between them.

1.2 The Architecture

The basic design principles of the architecture are as follows:

1. The architecture should make no assumptions about the internal structure of components.
2. The architecture should allow components to be distributed or co-located.
3. Overall control flow and modality coordination should be separated from user interaction.
4. The various modalities should be independent of each other. In particular, adding a new modality should not require changes to any existing ones.
5. The architecture should make no assumptions about how and when modalities will be combined.

The third and fourth principles motivate the most basic features of the design. In particular item 3 requires that there be a separate control module that is responsible for coordination among the modalities. The individual modalities will of course need their own internal control flow. For example, a VoiceXML-based speech

recognition component has its own internal logic to coordinate prompt playing, speech recognition, barge-in, and the collection of results. However the speech recognition component should not be attempting to control what is happening in the graphics component. Similarly the graphics component should be responsible for visual input and output, without concern for what is happening in the voice modality. The fourth point re-enforces this separation of responsibilities. If the speech component is controlling speech input and output only, while the graphics component is concerned with the GUI only, then it should be possible to add a haptic component without modifying either of the existing components.

The core idea of the architecture is thus to factor the system into an Interaction Manager (IM) and multiple Modality Components (MCs).

The Interaction Manager is responsible for control flow and coordination among the Modality Components. It does not interact with the user directly or handle media streams, but controls the user interaction by controlling the various MCs. If the user is using speech to fill in a graphical form, the IM would be responsible for starting the speech Modality Component and then taking the speech results from the speech MC and passing them to the graphics component. The IM is thus responsible for tracking the overall progress of the application, knowing what information has been gathered, and deciding what to do next, but it leaves the details of the interactions in the various modalities up to the MCs. A wide variety of languages can be used to implement Interaction Managers, but SCXML [5] is well suited to this task and was defined with this architecture in mind.

The Multimodal Architecture also defines an application-level Data Component which is logically separate from the Interaction Manager. The Data Component is intended to store application-level data, and the Interaction Manager is able to access it and update it. However the architecture does not define the interface between the Data Component and the IM, so in practice the IM will provide its own built-in Data Component.

Modality Components are responsible for interacting with the user. There are few requirements placed on Modality Components beyond this. In particular, the specification does not define what a "modality" is. A Modality Component may handle input or output or both. In general, it is possible to have "coarse-grained" Modality Components that combine multiple media that could be treated as separate modalities. For example, a VoiceXML-based Modality Component would offer both ASR and TTS capabilities, but it would also be possible to have one MC for ASR and another for TTS. Many Modality Components will have a scripting interface to allow developers to customize their behavior. VoiceXML is again a good example of this. However it is also possible to have hard-coded Modality Components whose behavior cannot be customized.

Note that to the extent that HTML5 is a multimodal language, it acts both as an Interaction Manager and as a Modality Component. The W3C Multimodal Architecture tries to provide more flexibility so that an application can use HTML5 as a graphical MC without being restricted to its limited control flow capabilities.

It is also possible to nest Modality Components. That is, multiple MCs plus an Interaction Manager can look like single MC to a higher-level Interaction Manager.

This design can be useful if some MCs need to be tightly coupled with each other, while they are more loosely coordinated with others. For example, ASR and TTS modalities are usually tightly coupled to coordinate prompt playing with recognition and barge-in. If a system was working directly with individual ASR and TTS Modality Components, it might want to couple them closely using a separate Interaction Manager. The resulting complex Modality Component would offer prompt and recognize capabilities to the larger application, similar to a native VoiceXML Modality Component.

In addition to the Interaction Manager and Modality Components, the architecture contains a Runtime Framework, which provides the infrastructure necessary to start and stop components, as well as enabling communication. The Runtime Framework contains a Transport Layer which must provide reliable, in-order delivery of events between the IM and the MCs. The Transport Layer might be HTTP for loosely coupled and distributed components or something proprietary for co-located and tightly coupled components. Overall, the Multimodal Architecture and Interfaces specification provides little detail on the Runtime Framework. However the separate Discovery and Registration specification [6] is filling in part of this gap.

Security is important for multimodal applications since they will often be dealing with sensitive information such as credit card numbers. However security is outside the scope of the Multimodal Architecture and Interfaces specification. The W3C Multimodal Working Group assumes that developers will consult the relevant security specifications when building their systems.

A diagram of the W3C Multimodal Architecture, taken from the specification [1], is given below (Fig. 1.1).

Overall, the W3C's Multimodal Architecture should look fairly familiar (lack of originality is considered a *good* thing in standards group work). One model for this design is the DARPA Hub Architecture, also known as the Galaxy Communicator [7]. The architecture can also be taken as an instance of the Model/View/Controller paradigm (especially when the data model is separate). Specifically, the Modality Components represent the view, while the Interaction Manager is the Controller.

The goal of this design is to chop the system up into pieces that are likely to make good re-usable components. For example, there are a number of open source SCXML interpreters that can be used as Interaction Managers. Similarly, an open source VoiceXML interpreter can be used as a Modality Component. On the other hand, the looseness of definition of Modality Components including the lack of a clear definition of "modality" is designed to allow room for experimentation while still providing enough structure so that the results of innovation can be re-used.

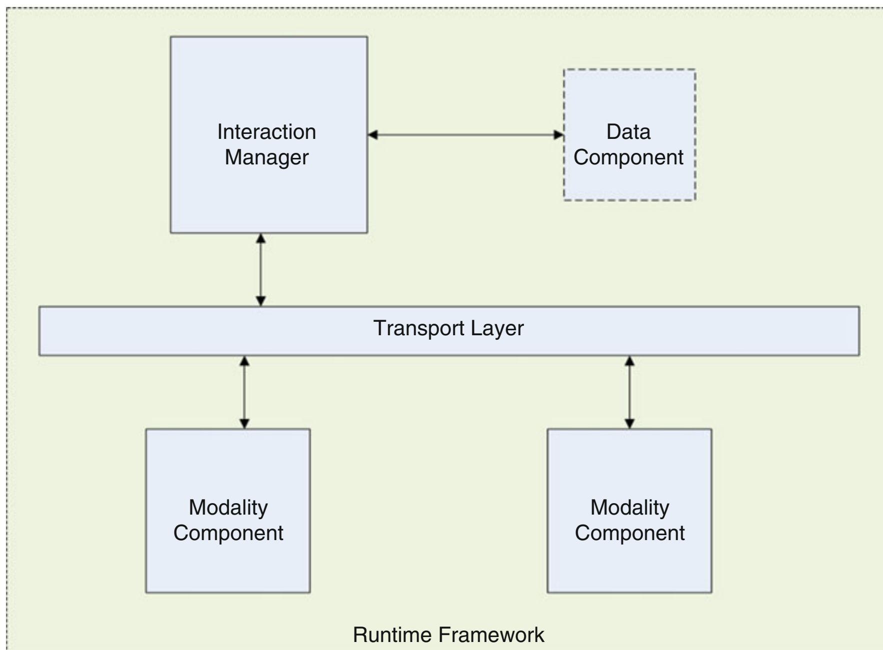

Fig. 1.1 Components of the multimodal architecture

1.3 The Interfaces

In addition to specifying the overall architecture, the Multimodal Architecture and Interfaces specification defines a set of events that are exchanged by the components. Since the specification does not commit to what the Modality Components are or to how they are implemented, the event set is high-level and generic, but still sufficient to build real-world applications. In keeping with the high-level nature of the event set, the events are defined as an abstract set of "fields," which any actual implementation would have to map onto a concrete syntax such as XML or JSON. The transport for the events is also not defined.

The majority of events are defined in request/response pairs. Certain fields are common to all events. The "target" field contains the address of the intended recipient, and allows the underlying messaging infrastructure to deliver the event. The "source" field gives the address of the sender, and the recipient of an event should be able to send a response to this address. The "context" field identifies a particular interaction with a user. For example, most VoiceXML and SCXML interpreters can handle multiple simultaneous sessions, so the "context" field allows the interpreters to determine which user session that event belongs to. The specification does not define the duration of a "context," but the Interaction Manager and all the Modality Components that are interacting with the user share the same "context," and it is possible for individual Modality Components to join and leave

the system during the lifetime of a "context." Finally a "RequestID" field allows components to match requests and responses, while the "data" field holds arbitrary data, and can be used to pass application-specific information.

Here is an overview of the event set:

- NewContextRequest/NewContextResponse. The first step in starting a new user interaction is creating a new context. If a Modality Component detects the presence of a new user, for example, by a phone call coming into a VoiceXML interpreter or a new visitor walking up to a multimodal kiosk, it can send the NewContextRequest event to the Interaction Manager. The IM will then respond by sending a NewContextResponse to the Modality Component containing a newly created context identifier. At this point all that has happened is a bit of book-keeping. The interaction with the user won't start until the IM sends a StartRequest (see below) to one or more Modality Components. The NewContextRequest is used when a Modality Component wants to create a new user interaction. The Interaction Manager can also create a new interaction at any point by sending a PrepareRequest or a StartRequest containing a new context identifier to one or more Modality Components. Thus a new user interaction may be started either by a Modality Component or by the Interaction Manager.
- StartRequest/StartResponse. The Interaction Manager starts a user interaction by sending a StartRequest to one or more Modality Components. The Modality Components return a StartResponse to acknowledge that they have begun running. The StartRequest contains two mutually exclusive fields, Content and ContentURL, that are used to instruct the Modality Component how it should interact with the user. It is most natural to think of these fields as specifying the markup that the Modality Component should execute. For a VoiceXML interpreter, for example, the Content field would contain an in-line specification of VoiceXML markup, while the ContentURL field would specify the URL to download the markup from. However Modality Components need not be controlled by markup. For ones that are not, the Content or ContentURL fields could contain platform-specific parameters or commands that would modify or control the behavior of the Modality Component. It is also possible to have a hard-coded Modality Component that runs the same way no matter what is specified in these fields.
- PrepareRequest/PrepareResponse. The PrepareRequest event is an optional event that the Interaction Manager can send before the StartRequest. It contains the same Content or ContentURL fields as the StartRequest. The purpose of the PrepareRequest is to allow a Modality Component to get ready to run by, e.g., downloading markup, compiling grammars, or any other operations that will allow it to start immediately upon receipt of the StartRequest. The PrepareRequest is useful for Modality Components that need to be tightly synchronized, for example, a text-to-speech engine that reads out a message while a graphical display highlights the words as they are spoken. If we simply send StartRequests to both components, it might take one longer to get going than the other, so coordination

will be smoother if the PrepareRequest allows both to prepare to start with minimal delay. The PrepareResponse is sent by the Modality Component back to the Interaction Manager to acknowledge the PrepareRequest.

- DoneNotification. This event is not part of a request/response pair, though it can be considered to be a delayed response to the Start Request. It is sent by a Modality Component to the Interaction Manager to indicate that it has finished its processing. For example, a text-to-speech system would send it when it had finished playing out the text specified in the StartRequest, or a VoiceXML interpreter would send a DoneNotification when it had finished executing its markup. (In this case, the VoiceXML interpreter might include an EMMA [8] representation of the recognition results in the event.) Not all Modality Components have a built-in concept of termination, so the DoneNotification is optional. For example, a simple graphical component might keep displaying the information that it had been told to display indefinitely until it received a new Start Request telling it to display different information. Such a component would never send a DoneNotification.
- PauseRequest/PauseResponse. The Interaction Manager may send a PauseRequest to a Modality Component, asking it to suspend its interactions with the user. The Modality Component then responds with a PauseResponse once it has paused. If a Modality Component is unable to pause, it will send a PauseResponse containing an error code.
- ResumeRequest/ResumeResponse. The Interaction Manager may send a ResumeRequest to any Modality Component that it has previously paused. The Modality Component will return a ResumeResponse once it has resumed processing. It is an error for the Interaction Manager to send a ResumeResponse to a Modality Component that has not previously been paused.
- CancelRequest/CancelResponse. The Interaction Manager may send a CancelRequest to any Modality Component telling it to stop running. The Modality Component will stop collecting user input and return a CancelResponse.
- ExtensionNotification. This event is intended to carry application- or platform-specific logic. Either the Interaction Manager or the Modality Components may send it, and no response is required (though the recipient could reply with another ExtensionNotification). This event includes a "name" field, which holds the name of the application- or platform-specific event, as well as an optional "data" field, which can hold an application- or platform-specific payload. A re-usable component, whether an Interaction Manager or a Modality Component, should document the set of ExtensionNotifications that it expects to send and receive, as well as their semantics.
- ClearContextRequest/ClearContextResponse. The Interaction Manager can send a ClearContextRequest to a Modality Component to indicate that the particular context/interaction is finished. The Modality Component is not required to take any specific action in response to this event, but normally it would free up any resources it has allocated to the interaction (cached grammars, adapted voice models, etc.) The Modality Component then responds with a ClearContextResponse.

- StatusRequest/StatusResponse. This event may be sent by either the Interaction Manager or a Modality Component, and is intended to provide keep-alive functionality. The recipient will reply with the StatusResponse event with a "status" field containing "alive" or "dead." (If the recipient doesn't respond at all, it is obviously dead.) The "context" field in the StatusRequest event is optional. If it is present, the recipient will reply with the status of that specific context/interaction. (A status of "alive" means that the context is still active and can receive further events.) If the "context" field is absent, the recipient replies with the status of the underlying server. In this case, a status of "alive" means that the server is able to process new contexts/interactions.

1.4 Some Examples

As an example of how this event set could be used in practice, consider a simple application running on a hand-held device consisting of a form that can be filled out either by speech or by typing in the values of fields in the GUI. This application would consist of a GUI Modality Component, a Speech Modality Component, and the Interaction Manager. The Modality Components gather the values of the fields and return them to the Interaction Manager, which will process then and submit the form when it is complete.

The event flow for starting the application and filling out a single field by speech would be as follows:

1. The IM sends a StartRequest event to the GUI MC.
2. The GUI MC displays the form and returns a StartResponse event.
3. The user selects a field by tapping on it. The GUI MC sends an ExtensionNotification with the name of the field to IM.
4. The IM sends a StartRequest to the Speech MC. The selected field will be specified in-line in the "Content" field or via a URL in the "ContentURL" field.
5. The Speech MC starts listening for speech and sends a Start Response.
6. The user speaks the value of the field. The Speech MC returns a DoneNotification containing the recognition result.
7. The IM sends an ExtensionNotification to the GUI MC specifying the value of the field (taken from the DoneNotification). The GUI MC updates its display with this value (Fig. 1.2).

If it takes the Speech Modality Component an appreciable amount of time to load and compile its grammars, and response time is a concern, the application could be modified so that the Interaction Manager would send multiple PrepareRequests to the Speech MC at start-up time, allowing it to prepare its grammars before the GUI MC displayed the form. In this case, the Speech MC would send a separate PrepareResponse for each request, and the IM would wait for all the responses before sending the StartRequest to the GUI MC. Events 1–7 would then occur in the same order as shown above.

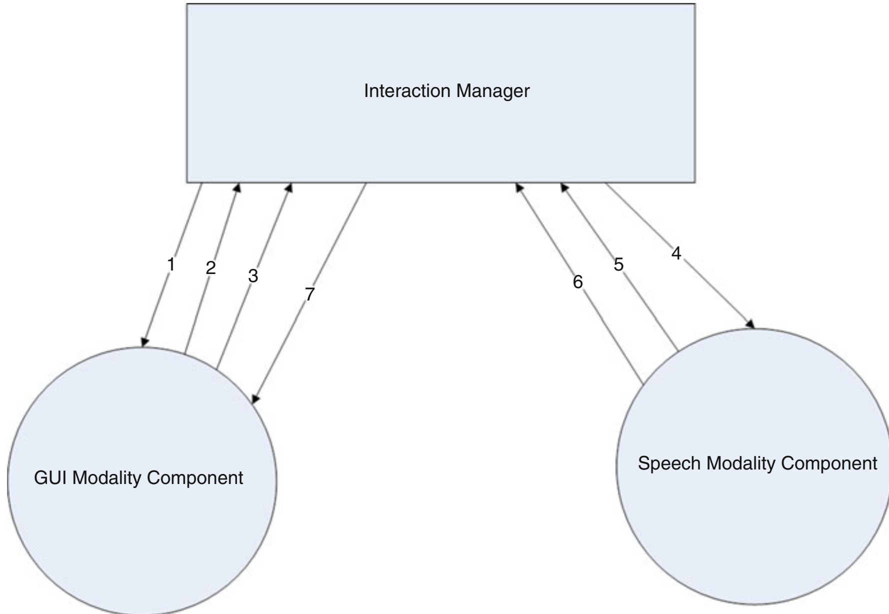

Fig. 1.2 Event sequence for filling a single field by voice

Now suppose that the user types in the value rather than speaking it. Events 1–5 remain the same, but this time it is the GUI MC that returns the value to the IM. The resulting event flow is as follows:

1. The IM sends a StartRequest event to the GUI MC.
2. The GUI MC displays the form and returns a StartResponse event.
3. The user selects a field by tapping on it. The GUI MC sends an ExtensionNotification with the name of the field to IM.
4. The IM sends a StartRequest to the Speech MC. The grammar for the selected field will be specified in-line in the "Content" field or via URL in the "ContentURL" field.
5. The Speech MC starts listening for speech and sends a Start Response.
6. The user types the value of the field. The GUI MC returns an ExtensionNotifcation containing the value. (Unlike the Speech MC, the GUI MC does not return values in a DoneNotification because it will keep running—that is, displaying the form—after it returns the result.)
7. The IM sends a CancelRequest to the Speech MC.
8. The Speech MC stops listening for speech and returns a CancelResponse (Fig. 1.3).

Since the user is using two modalities, there is a possibility of conflict, for example, if the user types one value and speaks another for a given field. It is the Interaction Manager's job to resolve such problems. It should keep its own Data

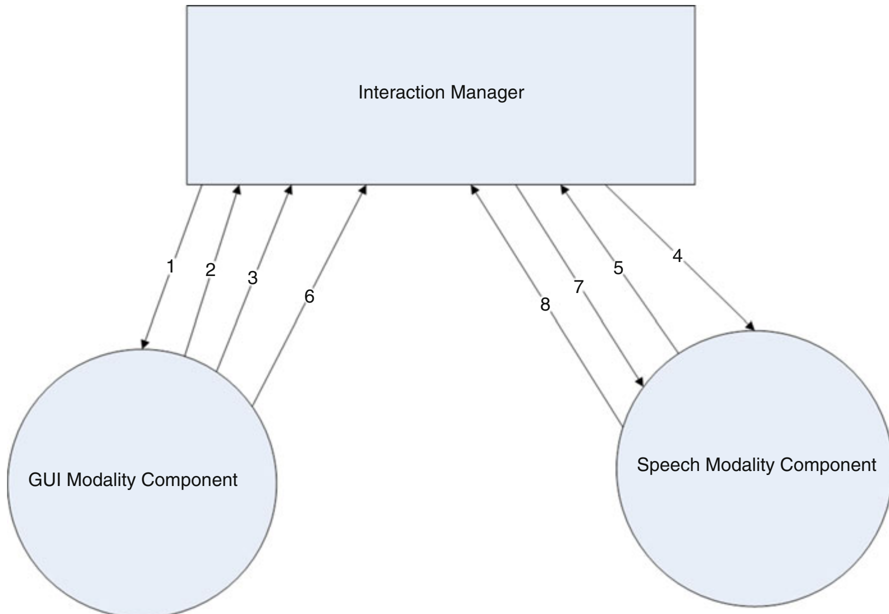

Fig. 1.3 Event sequence for filling a single field via the GUI

Component updated with the current state of the form. If a Modality Component sends the IM a value for a field that already has a value in the IM's Data Component, the IM knows a conflict has arisen. It is up to the application developer to decide what heuristic to use to resolve such conflicts since the Multimodal Architecture and Interfaces specification does not attempt to incorporate or enforce any particular approach to user interface development.

It is possible to modify the application so that the speech recognition doesn't follow the GUI field by field. Given a Modality Component that supports VoiceXML, the Interaction Manager can send it a StartRequest with a VoiceXML script that can capture the entire form. The VoiceXML Modality Component will now prompt the user for the various fields and gather input independent of what the GUI is doing. In fact, the user can speak the value for one field while simultaneously typing in the value of another. A sample event flow for such an application is given below:

1. The IM sends a StartRequest to the GUI MC.
2. The IM sends a StartRequest to the VoiceXML MC, either specifying the VoiceXML script in-line via the "Content" field, or by URL via the "ContentURL" field. (This event could also have been sent before the StartRequest to the GUI MC.)
3. The GUI MC displays the form and returns a StartResponse. (Depending on the timing of the application, this event could arrive at the IM before it sends the StartRequest to the VoiceXML MC.)

4. The VoiceXML MC loads the VoiceXML script and starts prompting the user for input. It then sends a StartResponse to the IM.
5. The VoiceXML MC obtains the value of one field and returns it to the IM in an ExtensionNotification Event.
6. The IM notifies the GUI MC of the field value with an ExtensionNotification event. The GUI MC updates its display accordingly.
7. The GUI MC obtains the value for a field and notifies the IM of it with an ExtensionNotification event.
8. The IM sends the field value to the VoiceXML MC in an ExtensionNotification event. If the VoiceXML MC updates its internal data model with this value, the Form Interpretation Algorithm [9] will ensure that it does not prompt the user for the value of this field.

   . . ... the user continues filling out the form using both modalities. . ...

9. The VoiceXML MC obtains the value for the final field and returns it to the IM in a DoneNotification.
10. The IM sends an ExtensionRequest to the GUI MC with the final value. The GUI MC updates its display (Fig. 1.4).

At the end of the event sequence shown above (i.e., after event ten reaches the GUI MC), the GUI MC is displaying the completed form, and the VoiceXML MC has stopped running. It is up to the application developer what happens next. The Interaction Manager will presumably submit or save the form. If more information

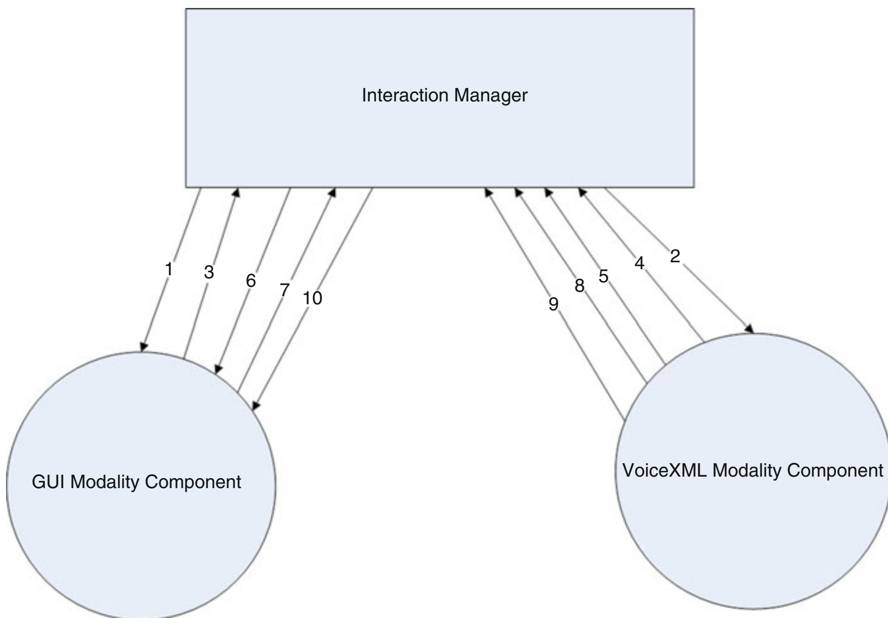

Fig. 1.4 Event sequence for filling multiple fields with voice and GUI

needs to be gathered, the Interaction Manager could send new StartRequests to both the GUI MC and the VoiceXML MC to continue the interaction with the user. One subtlety to note is the importance of the ContextID. If the new StartRequests contain the same ContextID as those in events 1–10, the Modality Components will view these requests as a continuation of the earlier interaction. Thus both Modality Components will keep any user adaptation they have performed. For example, the VoiceXML Modality Component will keep any speaker adaptation it has done to its voice models, while the GUI Modality Component will keep any display adjustments or special fonts that the user has selected. On the other hand, if the Interaction Manager sends ClearContextRequests before the StartRequests or simply uses a new ContextID in the StartRequests, the Modality Components will treat the requests as the start of a new interaction, possibly with a new user.

1.5 Adding a New Modality Component

As is clear from these examples, Modality Components do not communicate directly with each other, but only with the Interaction Manager. The value of this loose coupling becomes clear when a new Modality Component is introduced. Suppose the application is extended with a tablet capable of performing handwriting recognition. This Handwriting Modality Component will also communicate only with the Interaction Manager, sending it results via ExtensionNotifications. Neither the GUI nor the Speech Components need to be modified to work with the Handwriting Modality Component and their communication with the Interaction Manager will not change. The event flow for the user entering a field value with handwriting is shown:

1. The user taps on a field to select it. The GUI MC sends an ExtensionNotification to the IM telling it which field has been selected.
2. The IM sends the StartRequest to the Speech MC.
3. The Speech MC starts recognizing and returns a StartResponse.
4. The user writes out the value of the field using a stylus. The Handwriting MC sends this result back to the IM in an ExtensionNotification.
5. The IM sends an ExtensionNotification to the GUI MC containing the value for the field. The GUI updates its display.
6. The IM sends a CancelNotification to the Speech MC.
7. The Speech MC stops listening for speech and sends a CancelReponse. (The IM could just as easily have cancelled the Speech MC before notifying the GUI MC.) (Fig. 1.5)

Comparing this example to the first and second ones, it is clear that when the user enters a value via handwriting, the Speech MC receives the same events as when the user entered the value via the GUI MC. Similarly, the GUI MC receives the same event as when the user entered the value with speech. In fact, each Modality Component knows only that some other component has provided a value for the

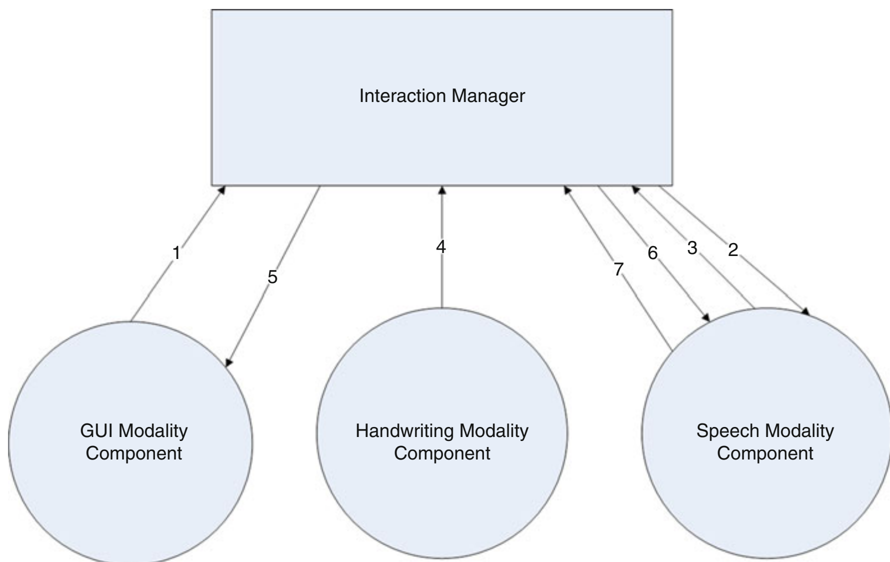

Fig. 1.5 Event sequence with handwriting component added

field in question. Only the Interaction Manager is aware of the new Modality Component. It is clear from this that the key to a successful implementation of the Multimodal Architecture and Interfaces specification is a powerful and flexible Interaction Manager, particularly one with good event handling capabilities. Given such an Interaction Manager, the design of the individual Modality Components is significantly simplified.

One important feature of the examples is the prevalence of ExtensionNotification events. The Multimodal Working Group felt that it was too early to define specific interfaces to modalities, with the result that ExtensionNotification carries a lot of the modality-specific logic. A Modality Component that supports this architecture will likely specify a lot of its interface in terms of ExtensionNotifications. For example, the GUI Modality Component's API specification might say that it will update the value of a field upon receipt of an ExtensionNotification event with name="fieldValue" and data="fieldname=value." One reason SCXML is a good candidate for an Interaction Manager language is that it has the ability to send and receive events with a variety of payloads, so that an SCXML interpreter can work with different Modality Components without requiring additional coding or integration work, particularly if the Modality Components support HTTP as an event transport.

1.6 Conclusion

The W3C's Multimodal Architecture is far from the last word on the subject. It is intended as an initial framework to allow cooperation and experimentation. As developers gain experience with this framework, the W3C Multimodal Working can modify it, extend it, or replace it altogether if a superior alternative emerges. The event set is quite high-level and will undoubtedly need to be refined if it becomes widely used. One obvious step would be to require support for a specific event syntax and transport protocol (for example, XML over HTTP). This would obviously facilitate interoperability and the only reason the Multimodal Working Group did not include such a requirement in the specification was the lack of consensus on what the syntax and transport protocol should be.

As another possibility for refinement, notice how often ExtensionNotifications are used in the examples given above to tell a Modality Component to update its internal data model. Perhaps an UpdateData event would prove useful. A further possibility would be to add modality-specific events. For example, if consensus emerges on how to manage a speech recognition system in a multimodal context, then a speech-specific event set could be defined.

Similarly, the multimodal architecture is quite high-level and will need to be articulated further. One possibility would be to add an Input Fusion component to the Interaction Manager. Consider the case of a user who says "I want to go here" and clicks on map. The utterance "I want to go here" will be returned by the speech Modality Component while the click will be captured by a graphical Modality Component. To understand the utterance, the system must combine the input from the two modalities and resolve "here" to the location on the map that the user clicked on. Right now this sort of combination is one of the many responsibilities of the Interaction Manager, but it might make sense to have a component that specialized in this task. Such a component would also be responsible for resolving conflicts between the voice and graphics Modality Components that were noted in the examples above, namely when the user types and speaks a value for the same field at the same time. See [10] in this volume for a more detailed discussion of how such a component might work. The Multimodal Working Group considered adding an Input Fusion component to the architecture, but decided that it didn't make sense to try to standardize the interface to such a component when there was not good enough agreement at the time about how it should work.

Finally, it is clear that many existing languages aren't easy to use as Modality Components because they don't allow fine-grained control. Both HTML and VoiceXML are designed to be complete stand-alone interfaces and it is not easy for an external component like an Interaction Manager to instruct a web browser what part of a page to display, or to tell a running VoiceXML interpreter to pause jump to another part of a form. Modality Component languages will fit into the W3C multimodal architecture much more easily if they are designed to accept asynchronous updates to both their data models and their flow of control.

References

1. Barnett, J., Bodell, M., Dahl, D., Kliche, I., Larson, J., Porter, B., et al. (2012). Multimodal architecture and interfaces. W3C Recommendation. http://www.w3.org/TR/mmi-arch/.
2. SALT Forum (2002). Speech Application Language Tags. http://xml.coverpages.org/SALT-FinalSpecificationV10.zip.
3. Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Navarra, E., O'Connor, E., et al. (2014). HTML5. W3C Recommendation. https://www.w3.org/TR/html5/.
4. Oshry, M., Auburn, R., Baggia, P., Bodell, M., Burke, D., Burnett, D., et al. (2007). Voice Extensible Markup Language (VoiceXML) 2.1. W3C Recommendation. https://www.w3.org/TR/2007/REC-voicexml21-20070619/.
5. Barnett, J., Akolkar, R., Auburn, R., Bodell, M., Burnett, D., Carter, J., et al. (2015). State Chart XML (SCXML) State Machine Notation for Control Abstraction. W3C Recommendation. https://www.w3.org/TR/scxml/.
6. Rodriguez, B. H., Barnett, J., Dahl, D., Tumuluri, R., Kharidi, N., & Ashimura, K. (2015). Discovery and registration of multimodal modality components: State handling. W3C Working Draft. https://www.w3.org/TR/mmi-mc-discovery/.
7. Galaxy Communicator (2003). http://communicator.sourceforge.net/sites/MITRE/distributions/GalaxyCommunicator/docs/manual/.
8. Johnston, M., Baggia, P., Burnett, D., Carter, J., Dahl, D., McCobb, G., et al. (2009). EMMA: Extensible MultiModal Annotation markup language. W3C Recommendation. http://www.w3.org/TR/2009/REC-emma-20090210/.
9. McGlashan, S., Burnett, D., Carter, J., Danielsen, P., Ferrans, J., Hunt, A., et al. (2004). Voice Extensible Markup Language (VXML) Version 2.0. Appendix C. W3C Recommendation. https://www.w3.org/TR/voicexml20/#dmlAFIA.
10. Schnelle-Walka, D., Duarte, C., & Radomski, S. (2016). Multimodal fusion and fission within the MMI architectural pattern. In D. Dahl (Ed.), *Multimodal Interaction with W3C Standards: Toward Natural User Interfaces to Everything*. New York, NY: Springer.