IrisTK: a Statechart-based Toolkit for Multi-party Face-to-face Interaction

Gabriel Skantze
KTH Speech Music and Hearing
Stockholm, Sweden
gabriel@speech.kth.se

Samer Al Moubayed
KTH Speech Music and Hearing
Stockholm, Sweden
sameram@kth.se

ABSTRACT
In this paper, we present IrisTK - a toolkit for rapid development of real-time systems for multi-party face-to-face interaction. The toolkit consists of a message passing system, a set of modules for multi-modal input and output, and a dialog authoring language based on the notion of statecharts. The toolkit has been applied to a large scale study in a public museum setting, where the back-projected robot head Furhat interacted with the visitors in multi-party dialog.

Categories and Subject Descriptors
H.5.2 [**Information Interfaces and Presentation**] : User Interfaces – Natural Language; D.2.2 [**Software Engineering**] Design Tools and Techniques – State diagrams; D.2.11 [**Software Engineering**] Software Architectures – Languages

Keywords
Multi-party interation, Gaze, Gesture, Speech, Spoken dialog, Multimodal systems, Statecharts

1. INTRODUCTION
Spoken dialog technology has for a long time focused on the linguistic interaction, and not as much on the physical space where the interaction takes place. Applications such as ticket booking over the telephone assume that the system interacts with a single user, and that the system and user are not physically co-located. While embodied conversational agents (ECAs) assume some sort of co-location, the ECA is still located behind a display, and the user's physical location in relation to the agent is most often not taken into account. However, there are many recent efforts at locating the interaction in a physical space which is shared between system agent and the users. An example of this is [1], where a virtual receptionist is presented which interacts with several users in a multi-party dialog. The animated agent is still presented on a screen, but the system uses visual input to detect the users' attentional states, and direct the system's attention in order to regulate the interaction. A related field of research is human-robot interaction, where the physical space in which the interaction takes place is of great importance [2].

ICMI '12, October 22–26, 2012, Santa Monica, California, USA.

One enabling factor behind this development is the advances in visual processing techniques (such as face detection and body tracking) that are needed to fully model face-to-face interaction, and the cheap commercial products this has resulted in, such as Microsoft Kinect[1]. This allows a wide audience of researchers and developers to experiment with multimodal interfaces. However, the combination of all these techniques together with spoken dialog technology is not trivial. Face-to-face interaction involves a large amount of real-time events that need to be orchestrated in order to handle phenomena such as overlaps, interruptions, coordination of headpose and gaze in turn-taking, etc. Also, the knowledge to develop and put together all necessary modules is of a very interdisciplinary nature. This calls for a dialog system toolkit for multi-party face-to-face interaction, which provides necessary modules for multimodal input and output and allows the developer or researcher to author the dialog flow in a way that is simple to understand for the novice, yet powerful enough to model more sophisticated behaviors. Such a toolkit should define clear interfaces to make it easy for a researcher who is interested in a specific aspect (such as the use of headpose and gaze in interaction) to adjust specific modules and easily adapt the dialog flow, without being an expert on dialog systems or body tracking. In this paper we present an effort towards developing such a toolkit, called *IrisTK*.

We will start the exposition by describing a challenging setting where IrisTK has been applied: a museum setup where the robot head Furhat interacted with the visitors in multi-party dialog. Over four days the exhibition was seen by thousands of visitors, resulting in a corpus of about 10.000 user utterances. This real-life complex example not only indicates the stability of the toolkit, but also helps us to exemplify many of its features. We will then continue by describing the general principles of the events that are sent in the system and the modules that were used in the exhibition. After this, we will describe the dialog authoring language *IrisFlow* that has been developed as part of the toolkit. We end with a comparison with related work and a discussion on our current and future work.

2. FURHAT AT ROBOTVILLE
The development of IrisTK is part of the IURO project[2], which aims at exploring how robots can be endowed with capabilities for obtaining missing information from humans through spoken interaction. The test scenario for the project is to build a robot that can autonomously navigate in a real urban environment, approach crowds of pedestrians, and enquire them for route directions. In December 2011, the IURO project was invited to take part in the Robotville exhibition at the London Science Museum, showcasing some of the most advanced robots currently being developed in

[1] http://kinectforwindows.org/
[2] http://www.iuro-project.eu/

Europe. In order to explore how a robot can gather information from humans through multi-party dialog, we put the interactive robot head Furhat [3] on display. During the four days of the exhibition, Furhat's task was to collect information on peoples' beliefs about the future of robots, in the form of a survey.

There are several examples of multimodal dialog systems put to the test in public settings [4,5,6]. Allowing spoken interaction in a public setting is indeed a very challenging task – the system needs to cope with a lot of noise and crowds of people wanting to interact at the same time. To make the task feasible, different restrictions are often applied. One example is the virtual museum guide Max [5], which only allowed written input. Another example is the museum guides Ada and Grace [6], which did not allow the visitors to talk to the agents directly, but instead used a "handler" who spoke to the system, that is, a person who knew the limitations of the system and "translated" the visitors' questions. Also, in that system, the dialogue was very simplistic – basically a mapping of questions to answers independent of any dialog context. What makes the Furhat at Robotville exhibition special, apart from allowing the visitors to talk directly to the system, is that the visitors interacted with the system in a multi-party dialog, allowing several visitors to talk to the system at the same time. While there are examples of systems that have engaged in multi-party dialogue in more controlled settings, such as the virtual receptionist presented in [1], we are not aware of any other multi-party dialogue system put to the test in a public setting, interacting with a large number of users.

Most studies on more sophisticated multi-party interaction (such as [1]) have utilized an embodied conversational agent presented on a flat screen. This may be problematic, however, due to the phenomenon known as the Mona Lisa effect [7]: Since the agent is not spatially co-present with the user, it is impossible to establish exclusive mutual gaze with one of the observers – either all observers will perceive the agent as looking at them, or no one will. While mechanical robot heads are indeed spatially and physically co-present with the user, they are expensive to build, inflexible and potentially noisy. The robot head Furhat that we have used here [3], shown in Figure 1, can be regarded as a middle-ground between a mechanical robot head and animated agents. Using a micro projector, the facial animation is projected on a three-dimensional mask that is a 3D printout of the same head model used in the animation software. The head is then mounted on a neck (a pan-tilt unit), which allows the use of both headpose and gaze to direct attention. It has previously been shown in an experimental setting that such a 3D projection increases the system's ability to regulate turn-taking in multi-party dialog, as compared to a 2D screen [8].

The setting of a public exhibition in a museum poses considerable challenges to a multimodal dialog system. In order to engage in a multi-party, situated interaction, the system not only needs to cope with the extremely noisy environment, but also be able to sense when visitors are present. For robustness reasons, we used two handheld close-range microphones put on podiums with short leads, forcing visitors to walk up to one of the microphones whenever they wanted to speak to Furhat. To sense whether someone was standing close to a microphone, ultrasound proximity sensors were mounted on the podiums. Furhat and the two podiums formed an equilateral triangle with sides of about 1.5 meter. On the wall next to Furhat, a screen was mounted with charts showing the real-time results of the survey. The purpose of this was to make the whole exhibition more interesting for the visitors. A snapshot of the setup can be seen in Figure 2.

An example dialog is shown in Table 1, which illustrates a number of typical interaction patterns. As soon as Furhat was approached by a visitor, Furhat immediately took the initiative and started to ask questions, as can be seen in turn 1-4. When the system understood an answer, it gave some relevant feedback (as in turn 6), but if it did not understand, it simply continued (as in turn 9 and 17). All answers were recorded and information about the corresponding questions was logged, which made it possible to annotate all answers later on. After each question, the system also made an *elaboration request* (as in turn 6 and 15). All utterances from the system (including questions) were randomly selected from a set of possible utterances, resulting in a varied output.

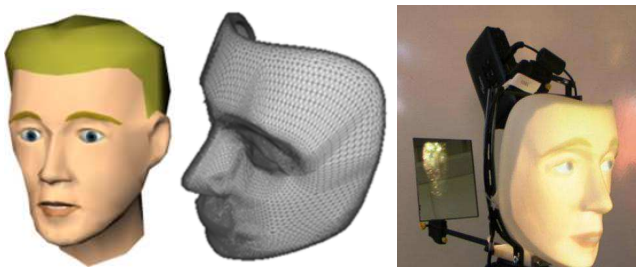

Figure 1: The 3D animation model used for Furhat (left), the mask (middle) and the projection (right).

Figure 2: Furhat at Robotville.

With two users present, Furhat could either ask a *directed question* – with the head posed in direction towards the addressee, and eyes looking forward (establishing eye-contact) – or an *open question* to both of them – with the head directed between the users, while alternating gaze between them (as in turn 17). Furhat then turned to the person who answered the question. When speech was detected in both microphones at the same time, the audio levels were compared in order to choose who to attend to. If a question was directed to one of the users and the other user tried to take the turn, Furhat would acknowledge this by shifting the gaze towards this user and say something like "could you please wait a second", while keeping the head directed towards the original user. Furhat would then shift the gaze back and continue with the interaction he was previously involved with (as in turn 11). Another option for Furhat in the multi-party setting was to follow up an elaboration request with an *agreement request*, by turning to the other interlocutor (as in turn 13). This could sometimes result in longer sequences on the same topic, such as the one shown in turn 9-16: question-elaboration-agreement-elaboration.

Although Furhat always started the interactions by asking questions from the survey, it was also possible for the visitors to ask questions (as in turn 18) and thereby shift the topic for the dialog. This would trigger Furhat into awaiting further questions, resulting in mixed-initiative interaction patterns. After answering some questions, or after too many non-understandings, Furhat again tried to get the initiative and ask questions from the survey.

Over the four days of the exhibition, the system ran continuously without any problems, which indicates the robustness of the toolkit. This setup allowed us to collect a corpus of about 10.000 user utterances and to perform large-scale experiments in which we could investigate how people respond to a survey robot and how the 3D design of Furhat allows for accurate turn-taking regulation in a multi-party setting [9]. We also selected 86 visitors who actively interacted with the robotic head, and asked them to fill in a short questionnaire on their impression of the conversation, using a 5-point Likert scale. The participants' overall impression of the system was very positive. The results indicated that they liked Furhat (mean = 4.08, SD .76), they enjoyed talking to the robot (mean = 4.13, SD .84), and they liked Furhat's response behavior (mean = 3.80, SD .71). For a more detailed analysis, see [10].

Table 1: Example interaction. ASR results shown in brackets, using S: SRGS or D: Dictation grammar.

1. U1	*(Approaching microphone 1)*
2. F	*(Looking up at U1)* Hi there. Could you perhaps help me?
3. U1	Yes [S: **yes**]
4. F	I have some questions for you. When do you think robots will beat humans in football?
5. U1	Maybe in 10 years [D: that see **in 10 years**]
6. F	That soon! Could you elaborate on that?
7. U1	Well, they are already pretty good at it [D: while they are already predicted owners]
8. U2	*(Approaching microphone 2)*
9. F	*(Still looking at U1)* Yeah… I have another question for you. Would you like robots to be like humans?
10. U2	Yes, absolutely [D: **yes** that see]
11. F	*(Turning to U2 with gaze only)* Could you just wait a second? *(Looking back at U1)* I'm sorry, where were we. Would you like robots to be like humans?
12. U1	Sure [S: **sure**]
13. F	Yeah? *(Turning the head towards User-2)* Do you agree?
14. U2	No [S: **no**]
15. F	Mhm. Could you say something more about that?
16. U2	Well, I want robots to be like robots [D: I want robots to be like Reynolds]
17. F	Yeah… *(Posing head between users, shifting gaze back and forth between them)* I have another question for you. When do you think robots will serve hamburgers at McDonalds?
18. U1	Do I need an umbrella? [S: **do I need an umbrella**]
19. F	*(Turning to U1)* What do you think, you are in London! Maybe you have another question for me?

3. ARCHITECTURE AND EVENTS

The system consists of a set of modules which produce and consume events. The three basic types of events are *action*, *sense* and *monitor*. *Action* represents things that the system should do, such as saying something, producing a gesture or turning the head. *Sense* represents things that the system perceives (such as a user saying something or approaching Furhat). *Monitor* represents feedback sensations that modules are required to produce when performing an action. Such signals are essential to real-time systems, since they allow other modules to know when actions have been performed or are about to be performed, which facilitates synchronization, sequencing and interruptions of actions (for example if the utterance the system is producing needs to be interrupted because the user is leaving the interaction in the middle of it).

Figure 3 gives an abstract view of the general types of modules and events in the system. *Sensor* modules (such as cameras and microphones) produce low level sensations. Based on this, *interpreter* modules (parsing, gesture recognition, etc) produce high level sensations. The *controller* takes this high level sensation and produces a high level action. This is in turn broken down by *generator* modules (natural language generation, gesture synthesis, etc) into low level actions which are executed by *actuator* modules (motors, facial animation, etc). Of course, a controller does not have to operate on a high level; it may also directly map low level sensations to low level actions. It is also important to note that a single module can serve many of these roles at the same time, and that there can be several components that serve each role.

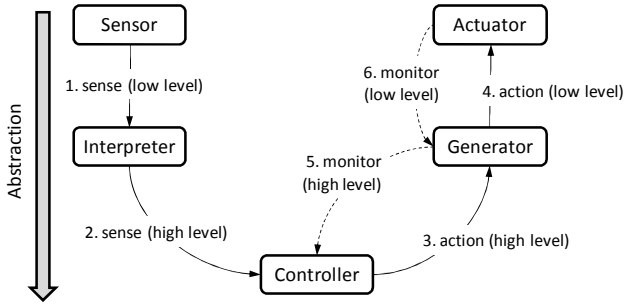

Figure 3: Types of modules and events.

The modules can run on the same computer in a single process or distributed over several processes and computers, using sockets to communicate, which allows for platform and programming language independence. All messages have a corresponding XML representation for serialization. By default, all messages that a module produces are sent to all other modules, but this can be changed in order to optimize the system.

4. EXAMPLE MODULES

We have implemented a range of modules for IrisTK that allows for face-to-face interaction. In this section we will describe these and the events that they produce, or may trigger them into action. A general principle behind all events is *symmetry* – a system might sense *speech* from a user, but also issue an action to produce *speech* as a response to this, which in turn will result in a *speech* monitor event by the module that realizes this action. This means that we represent speech from the system and speech from the users in a similar way.

4.1 Input: User locations

In order to participate in a multi-party face-to-face interaction, the system needs to be able to detect potential interlocutors. This includes both their locations and their identities. Identities might either be global (across sessions, for example by use of face recognition), or local (stable only during one session).We have explored different types of sensors for this, and incorporated them into IrisTK.

A simple and robust way of detecting the users' location, which was done for the Robotville exhibition, is to use off-the-shelf ultrasound proximity sensors. This requires that the possible locations are predefined. The proximity sensor can provide an estimate of the distance to the user, but not any hints to the user's identity.

A more advanced and flexible (but less robust) solution is to use Microsoft Kinect. Using a depth camera and skeletal tracking software, Kinect can track up to two people at the same time and provide the 3D coordinates of the body parts. Kinect can provide local identities of the bodies. We have implemented a module for the toolkit which uses Kinect to identify and track the users.

Another solution that we have used, which is more stable than Kinect and allows for more participants, is to use a regular camera and the face tracking software SHORE developed by Frauenhofer IIS [11]. The users are given wireless handheld microphones with colored markers. This allows the visual processing software to easily identify them (see Figure 4). If a face is detected close to the microphone, this indicates that a user is ready to talk to the system. It is also possible to detect if the users' heads are oriented towards the system agent (as an indication of their attentional state).

The basic event that is produced by these modules is `sense.body`. The message might include a predefined location (as is the case with the proximity sensor), or precise coordinates of the users, their body parts (head, hands, microphones, etc) and the rotation of the head, depending on the capabilities of the specific module.

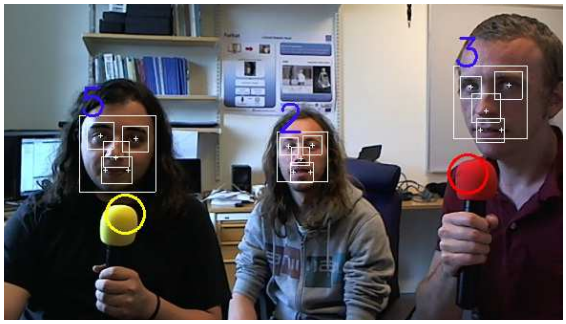

Figure 4: Microphone and face detection.

4.2 Input: Speech understanding

For speech recognition, we have been using the Windows 7 ASR. By running several ASR modules in parallel, it is possible to process speech from several microphones simultaneously (as was done in the Robotville exhibition). Each ASR engine can also use parallel language models: context-free grammars with semantic tags (SRGS[3]), tailored for the domain, and an open dictation model (as shown in Table 1). This makes it possible to handle "out-of-grammar" utterances to some extent. To interpret the dictation results, we have implemented a robust parser that uses the SRGS grammar to find islands of matching fragments (similar to [12]).

The speech recognition is started by issuing an `action.listen` event. It is possible to specify the location where the system should listen (which will only trigger the associated ASR module). It is also possible to re-weight the language models used, depending on the state of the interaction. At the start and end of speech, the speech recognizer produces `sense.speech` events. The end-of-speech event contains the recognized text and the semantic interpretation, as specified in the SRGS grammar, according to the SISR standard[4], which allows for a nestled key-value structure. All nodes in the result are associated with confidence scores.

[3] http://www.w3.org/TR/speech-grammar/
[4] http://www.w3.org/TR/semantic-interpretation/

Another possibility that we have explored is to use the microphone array in Microsoft Kinect, which allows the use of ASR without handheld microphones (at the cost of a lower ASR performance). The microphone array can also be used to determine the users' locations using built-in beam-forming algorithms.

4.3 Output: Furhat

To control Furhat (Figure 1) within IrisTK, two modules have been implemented, a *face* module and a *neck* module.

The face module handles the facial animation and speech synthesis. For speech synthesis, we use the CereVoice system developed by CereProc[5], lip-synchronized with the facial animation. The module listens for `action.speech` events, which includes the text to be said, annotated with SSML[6]. Feedback sensations are produced at the start and end of the utterance. The eye rotation (and thereby the gaze) of Furhat is controlled by `action.gaze` events. It is also possible to trigger predefined gestures in Furhat's face (such as smiling or raising the eyebrows) using `action.gesture` events. Currently, we only have a set of predefined gestures, but this could eventually be extended with a more powerful markup language, such as BML [13].

The neck module controls the pan-tilt unit and listens for `action.headpose` events. Both this event and the `action.gaze` event allows for duration to be set (in milliseconds). This makes it possible to coordinate different movements. For example, by moving the neck and gaze in opposite directions with the same duration parameters, Furhat can keep the gaze at a fixed target while moving the head.

4.4 Attention manager

The task of the Attention manager (AM) is to build a situation model and keep track of the users' locations and identities, and the attentional state of the system. Using this model, it can act as an interpreter and generator for events related to these issues, translating raw sensory input into higher level behaviors, and higher level action behavior into lower level actions. The AM also uses the situation model to assign id:s to the agents. Hence, the Dialog manager does not need to care about what sensory devices or actuators the system uses and can be used in different setups.

An example of how the AM and the other modules were used in the Robotville scenario is shown in Figure 5. This also illustrates the different roles that the different modules served (compare with Figure 3). As a user approaches the proximity sensor, a `sense.body` event is raised, which is then interpreted by the AM as someone entering a specific location (`sense.enter`). This in turn triggers the Dialog manager (the controller) to raise an `action.attend` event. This is caught by the AM, and is translated into `action.gaze` and `action.headpose` events. These are then consumed by the Neck and Face modules, making Furhat look up and shift the gaze towards the user. When these actions have been executed, a chain of feedback sensory events on different levels are raised (`monitor.gaze`, `monitor.headpose`, `monitor.attend`). This triggers the Dialog Manager to issue an `action.listen` event targeted towards the location where the user is standing. The ASR-1 module receives this and starts the recognizer, producing a `monitor.listen` event (as feedback) and `sense.speech` events (when the user starts and stops speaking).

[5] http://www.cereproc.com/
[6] http://www.w3.org/TR/speech-synthesis/.

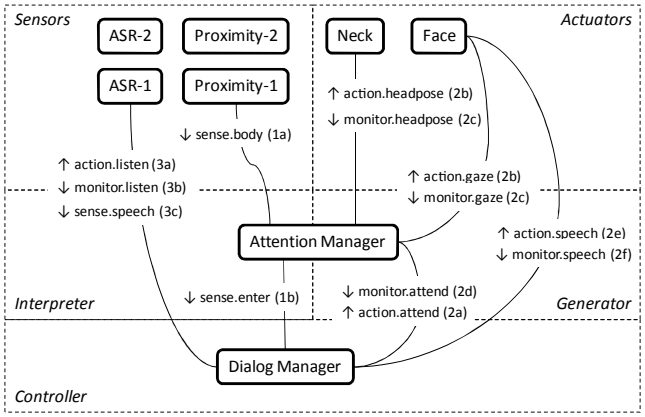

Figure 5: The flow of events between modules during the first three turns in Table 1. The numbers and letters represent the order in which the events are fired.

5. AUTHORING THE DIALOG FLOW

As can be seen in Figure 5, the central controller module (mapping sensations to actions) in the system is the Dialog manager. In this section, we will describe the formalism we use in IrisTK for authoring the behavior of the Dialog manager.

5.1 The statechart paradigm

The central problem of dialog management is that of selecting the next system action (in other words, mapping sensations to actions). The most straightforward approach is a direct mapping of the last user utterance to a system response, which may work for very simple question-answering systems [6]. However, since dialog is inherently context dependent, the selection of action often needs to depend on the current *dialog state*. Two common models which take the dialog state into account are finite state machines (FSM) and the information state update (ISU) approach [14]. FSMs are attractive in that they can be easily visualized and the flow is easy to understand. However, this only holds for simple models; the number of states and transitions can easily explode, and FSMs are therefore hard to use in, for example, mixed-initiative dialogs. In the ISU approach, the dialog state is represented as a collection of variables (the *information state*), such as the last user utterance, a stack of issues under discussion, etc. A set of rules are then applied that are conditioned on these variables and may cause other variables to change (causing other rules to apply) or result in actions that the system can take (similar to the concept of a *production system*). The ISU approach may scale better to more complex interaction patterns and domains than FSM, since the number of possible states the system can be in (in effect, all possible combinations of the variables) need not be enumerated (as is the case for FSMs). However, since these states are not explicitly defined, it may be hard to anticipate and get an overview of the system's behavior, as the number of variables and update rules grows. This also means that the state-space cannot be easily visualized, as is the case with (smaller) FSMs.

In [15], David Harel presented a powerful formalism for defining complex, reactive, event-driven systems, called *statecharts* (often referred to as Harel statecharts). When applied to dialog management, this paradigm can be regarded as a middle ground between ISU and FSM. Just like FSM, the dialog state is represented as a set of predefined states with transitions that are triggered by events. It is thereby possible for the designer to visualize the state-space and get an overview of the system's behavior. However, Harel added a number of extensions that drastically reduce the number of states and transitions needed. First, the states can be hierarchically structured; allowing the designer to define generic event handlers (i.e., transitions) on one level and more specific event handlers in the sub-states. Second, it is possible to add a *datamodel* (i.e., variables), which may affect further execution. Third, it is possible to add *guard conditions* to transitions, which are dependent on event parameters and the state of the datamodel. Fourth, transitions between states, or the entering or leaving of a state, can also be associated with *actions*, which can be used to raise internal or external events that either affect the further execution of the statechart, or give rise to actions in the other parts of the system (such as issuing the speech synthesizer to say something). Another extension that we will not discuss further in this paper is the support for concurrent states.

Statecharts has recently gained increasing interest as a model for dialog management. An XML-based representation which closely follows the original Harel statechart model, called SCXML[7], is currently being standardized by W3C. SCXML is planned to be part of the VoiceXML 3.0 standard[8], in order to allow for a more flexible control in telephone-based dialog systems. In the research community, it has been shown how SCXML can be used to model the ISU-approach to dialog management (using the *datamodel* to represent the information state) [16]. While this illustrates the flexibility of the formalism, we think that it is better to utilize the state space to directly model the different states of the dialog, in order to avoid the shortcoming of the ISU-approach discussed above.

5.2 IrisFlow – a statechart variant

While SCXML corresponds to the statechart model that Harel presented, we don't think the specification is optimal as an authoring language for dialog behavior. One of the problems that we have encountered is the handling of what Harel refers to as *activities*. These are defined as actions that take some time (i.e., that are not instantaneous), which obviously is very common for dialog systems (e.g., when the system says something or makes a gesture). In order to handle activities, the Harel statechart model assumes that each activity is associated with a state (i.e., the entering and leaving of the state are associated with the beginning and end of the activity). If an event handler needs to execute a sequence of activities (again, very common for dialog systems) and associate different event handlers with these, a new state has to be defined for each specific activity. Thus, it becomes cumbersome to manually author such behavior[9].

To make the authoring of the dialog behavior more efficient and simpler, we have defined an XML-based authoring language for dialog flow control, called *IrisFlow*, which can be regarded as a variant of Harel statecharts. The most important extension is the possibility of recursion (or context-freeness), which we think allows for a better handling of activities in dialog systems[10]. A single event handler can trigger a sequence of activities by a sequence of *parameterized state calls*. To allow this, we define two types of transitions: *goto* and *call*. While *goto* corresponds to a traditional statechart transition, *call* makes the execution model

[7] http://www.w3.org/TR/scxml/

[8] http://www.w3.org/TR/voicexml30/

[9] In VoiceXML 3.0, the individual dialog-related activities are not executed within SCXML. Instead, a complete VoiceXML interaction is associated with a single SCXML state. Instead, we want to control all dialog behavior within the statechart and not mix models.

[10] The possibility of such an extension is discussed in Harel's original article [15] (6.4).

put the current state and execution plan on a call stack before the transition takes place. The called state can then *return* to the calling state and execution plan without any explicit reference to a return point. Thus, the called state might be thought of as a *dynamic sub-state* (as opposed to the static sub-states defined by the state hierarchy).

5.3 IrisFlow example

A graphical representation of a small subset of the statechart used for the Robotville setup is shown in Figure 6. The boxing of the states illustrates the hierarchical structure. We do not show all possible transitions here, but instead the transitions that take place during the dialog in Table 1 (numbers on the arrows correspond to the different turn numbers in the scenario).

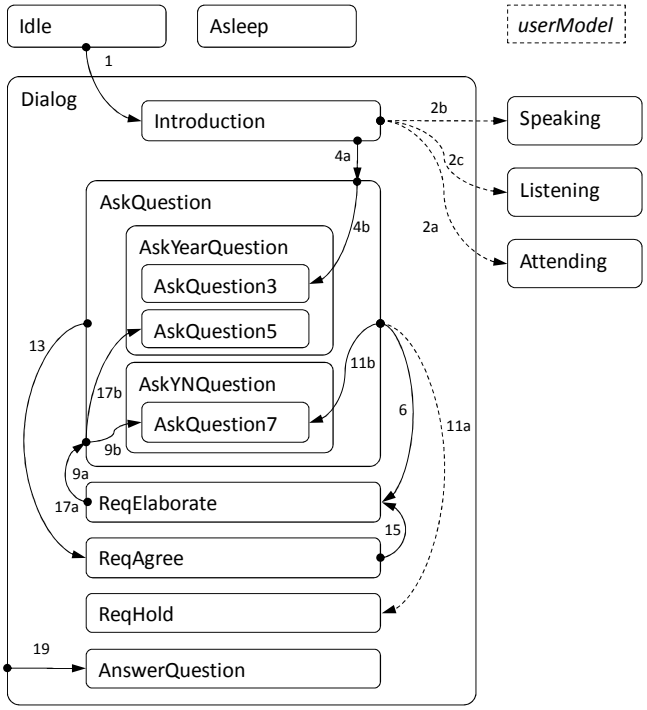

Figure 6: Fragment of the statechart for the Robotville exhibition. Arrows illustrate state transition for the example in Table 1 (solid lines=*goto*, dotted lines=*call*).

Using IrisFlow, we defined generic states, such as *Idle* and *Dialog*, with sub-states such as *Introduction* and *AskQuestion* (which in turn has sub-states such as *AskYNQuestion* and *AskYearQuestion*). The generic *Dialog* state defines event handlers to handle questions from the user regardless of the current sub-state, allowing mixed-initiative interaction. Dynamic sub-states were also defined, such as *Speaking*, *Attending* and *Listening*, with relevant event handlers. An event handler in the *Introduction* state can thus first call the *Speaking* state (asking the user a question) and then, when the speaking is completed, the *Listening* state (listening for the user's answer to the question). When in a called state (i.e., a dynamic sub-state), events are not only checked against the event handlers of the current state and its super-states, but also the event handlers of the calling states. Thus, when an event is processed and the flow is in the state *Speaking*, the event handlers of the following states will be checked (in order): *Speaking*, *Introduction*, *Dialog*. An important extension to Harel statecharts that makes the call transition useful to handle activities (that Harel also discussed in [15]; 6.1) is that transitions can be parameterized. For example, a called state can take parameters that are passed when the *call* is made (much like a function with arguments), which controls the behavior while in that state.

It is also possible to define variables on different level. As illustrated in Figure 6 (with dotted outline), a global variable called *UserModel* is also defined, which is used by the system to remember which questions it has previously asked a specific user, thus avoiding to ask the same question again. Each time the *AskQuestion* state is entered (see transition 4a, 9a and 17a in Figure 6), the UserModel is consulted to select which question should be asked next (resulting in transition 4b, 9b and 17b).

The IrisFlow statechart is authored using XML. The XML code is then compiled into Java source code, which is in turn compiled into Java byte code. Therefore, Java code can be used directly in the XML (in for example guard conditions), and there is no need for a scripting language such as JavaScript. An IrisFlow statechart (`<flow>`) contains a collection of states (`<state>`). A state, in turn, contains a collection of event handlers (`<catch>`). A small fragment of the definition of super-state *Dialog* is shown below. This defines generic event handlers that are used throughout the dialog, regardless of which specific dialog state the system may be in.

```
<state id="Dialog">
  <catch event="sense.leave"
         cond="userModel.attending(@agent)">
    <if cond=" userModel.present('loc-1')”>
      <call state="Attending" param:target="'loc-1'"/>
      <goto state="AskQuestion"/>
    <elseif cond=” userModel.present('loc-2')”/>
      <call state="Attending" param:target="'loc-2'"/>
      <goto state="AskQuestion"/>
    <else/>
      <goto state="Idle" param:goodbye="true" />
    </if>
  </catch>
  <catch event="sense.speech.end"
         cond="eq(@sem.act,'req_name')">
    <call state="Speaking"
          param:text="'My name is FurHat!'" />
    <goto state="AnswerNext" />
  </catch>
</state>
```

Each event handler (`<catch>`) corresponds to the `<transition>` element in SCXML, with possible guard conditions (`cond`). However, it only specifies a sequence of actions to take when an event is caught, and the actual transition is defined as an action (`<goto>`). As can be seen, both *call* and *goto* can be parameterized (attributes in the namespace *param*). The names of external events that can be caught (such as `sense.leave`) correspond to the name of the events described in section 4 above. It is also possible to raise and catch internal events which can be defined by the dialog author. As can be seen, the *Dialog* state defines the behavior when an agent leaves the interaction (turn to the other speaker if there is one, otherwise go to the Idle state). It also handles questions from the user (such as "what is your name?"). Below is a small fragment of the *Introduction* state, which extends the *Dialog* state[11]:

```
<state id="Introduction" extends="Dialog">
  <onentry>
    <random>
      <call state="Speaking"
        param:text="'Could you perhaps help me?'"/>
      <call state="Speaking"
```

[11] The <state> elements are not nestled as in SCXML, which we think makes it easier to distribute the code over several documents and potentially allows for "multiple inheritance" (again, an extension that Harel discuss in [15], 6.2).

```
        param:text="'Would you like to help me?'"/>
      <call state="Speaking"
        param:text="'Do you want to help me?'"/>
    </random>
    <call state="Listening"
          param:grammar="'default/0.5 yn/1.0'"/>
  </onentry>
  <catch event="sense.speech.end"
         cond="eq(@sem.act,'no')">
    <call state="Speaking" param:text="'Okay'"/>
    <goto state="Idle" param:goodbye="true"/>
  </catch>
  <catch event="sense.speech.end"
         cond="eq(@sem.act,'yes')">
    <call state="Speaking" param:text="'Great!'"/>
    <goto state="AskQuestion"/>
  </catch>
  <onidle>
    <call state="Speaking"
        param:text="'Sorry, I didn't hear you'"/>
    <reenter/>
  </onidle>
</state>
```

The *Introduction* state basically handles a simple yes/no question to the user. However, since it extends the *Dialog* state, it is still possible for the user to not answer the question but instead ask a counter-question, leave the interaction, etc., which allows for mixed-initiative. As in SCXML, the `<onentry>` is a special event handler that is triggered when the state is entered. The `<random>` element allows for a probabilistic behavior, which in this case allows for a more varied output. There is also a special event handler, `<onidle>` which is called when there are no more actions to take and no more events in the event queue (e.g., if the *Listening* call returns without triggering any event handlers). The `<reenter>` element causes the current state to be entered again.

Below is a small fragment of the *Listening* state, which is called when the system runs the speech recognizer (how the state is called can be seen in the *Introduction* state above):

```
<state id="Listening">
  <param name="timeout" default="8000"/>
  <param name="grammar" default="'default/1.0'"/>
  <onentry>
    <send event="action.gesture" param:name="'brow-up'"/>
    <send event="action.listen" param:timeout="timeout"
                                param:grammar="grammar"/>
  </onentry>
  <catch event="monitor.listen.end">
    <send event="action.gesture"
          param:name="'brow-down'"/>
    <return/>
  </catch>
</state>
```

As can be seen, the parameters that the state takes are explicitly defined with default values (`<param>`). On entry, the state issues an external event to start the recognizer and to raise the eyebrows, which signals to the user that the system is listening. When the speech recognizer ends (producing the feedback sensation `sense.listen`), the eyebrows are lowered again and the execution returns to where it left off in the calling state (`<return>`).

These examples show how the dialog behavior can be specified in a rather compact way. To implement the *Introduction* state above with SCXML, each call to the Speaking and Listening state would require their own state (a total of seven states instead of one), defined with their respective event handlers. Using IrisFlow, behavior that is specific to each activity can be defined using parameters, behavior that is specific to this collection of activities can be defined in *Introduction* state, and the general behavior when speaking can be defined in the *Speaking* state.

These examples only illustrate some basic principles; the complete IrisFlow XML used for Robotville contained 33 states (about 1600 lines of XML code).

6. DISCUSSION AND RELATED WORK

In the example presented here we only have one controller. It has been suggested [17,18] that it might be better to split the sensor-action mapping into modules that responsible for low-level actions (such as backchannels and engagement), and modules responsible for high-level actions (such as selecting the next dialog move). While we can see the benefits of such an approach when it comes to reusability of modules (e.g., the same backchannel or engagement module could be used in different applications), there is also a risk that it may result in conflicting actions being issued, especially as the complexity of the system increases. Also, since the purpose of IrisTK is to allow rapid development, all aspects of the behavior of the system should be easily authored, and it is therefore a great advantage if all control of the system actions are handled in a uniform way. We will therefore stick to a *single-controller architecture* as a best practice (although no such restrictions are enforced by the framework). When it comes to reusability, we think that the statechart-based design presented here provides an elegant solution. Super-states, such as *Dialog*, and dynamic sub-states, such as *Speaking* and *Listening*, can be defined for different kinds of dialogs (e.g., multi-party vs. dyadic) and for different setup of modules, and can then be reused across applications.

It should be noted that the example presented here is a practical system in a public setting. The states are very tightly coupled to the specific questions asked by the system, with hard-coded textual representations of the system's utterances. However, it is also possible to implement more generalized models of dialog, where the states correspond to more general discourse units such as Initiative and Response. It should also be possible to implement specific theories of dialogue, such as the Collaborative Discourse Theory [19], where the recursive state calls could be used to model Discourse Segments on different levels.

There are many other examples of research platforms for developing spoken dialog systems. One example is **TrindiKit** [14], which builds upon the ISU approach to dialog management discussed in 5.1. Another example is **RavenClaw** [20], which is based on the idea of automatically constructing a dialog model from a task model. **Jindigo** [21] is a framework targeted towards incremental dialog processing. **WAMI** [22] is a framework for multimodal systems that are to be run within a web-browser. None of these are especially targeted towards face-to-face or multiparty interaction; their primary focus is on the linguistic interaction. This does not necessarily mean that they cannot be used for this, but the frameworks do not provide useful modules for such interaction, nor have they been proven useful in such settings. More importantly, none of these frameworks are based on the statechart model for authoring the dialog flow, and are, in our opinion, not as intuitive and flexible when it comes to dialog modeling. There are of course more complex frameworks for multimodal robotic systems, such as **RT-Middleware**[12] and **ROS**[13]. While these frameworks are indeed mature and offer a range of modules, they do not have a special focus on the spoken interaction and do not provide any simple dialog authoring tools.

[12] http://www.openrtm.org/

[13] http://www.ros.org

The most well-known open standard for dialog flow authoring is perhaps **VoiceXML**. However, we do not think that VoiceXML can account for more sophisticated dialog behaviors, nor is there any support for multimodal events (beyond DTMF touchtones). While the upcoming version of the standard (3.0) has support for more flexible control, using **SCXML,** it is still unimodal. As discussed above, we also think that SCXML is less suitable for direct authoring of the dialog behavior.

7. FUTURE WORK

We are currently applying IrisTK to the final demonstrator in the IURO project: a robot that can autonomously navigate in a real urban environment, approach crowds of pedestrians, and enquire them for route directions. The statechart model has allowed us to define mixed-initiative dialog, where the system can first turn to one pedestrian and let her freely describe the route, then turn to another pedestrian and let him verify the route, segment by segment.

There are several directions in which IrisTK can be extended. Since the statechart paradigm lends itself well to visual representations, it should be possible to develop a graphical editor for designing the interaction flow, similar to the CSLU toolkit[14], but much more powerful. Another interesting extension is to allow the dialog designer to define explicit decision points in the flow (similar to the `<random>` statement), and associate rewards to specific states. Then, reinforcement learning could be used to automatically learn optimal decisions at these points when executing the statechart, by letting the system interact with real or simulated users [23].

We are also planning to release IrisTK as open source. Hopefully, this will be useful to other researchers and inspire them to contribute to the toolkit. While the focus in this paper has been on the benefits for multi-party face-to-face interaction, IrisTK can of course also be used for other kinds of dialog applications and experimental settings. To our knowledge, Furhat at Robotville is the first example of multi-party spoken interaction with a physically situated agent in a public setting. This was made possible by developing the toolkit that has been presented in this paper.

8. ACKNOWLEDGMENTS

This work is partly supported by the European Commission project IURO (Interactive Urban Robot), grant agreement no. 248314, as well as the SAVIR project (Situated Audio-Visual Interaction with Robots) funded by the Swedish Government (strategic research areas). The authors would like to thank Jonas Beskow, Joakim Gustafson, Björn Granström and Nicole Mirnig for their contribution to the Robotville exhibition.

9. REFERENCES

[1] Bohus, D., & Horvitz, E. (2010). Facilitating multiparty dialog with gaze, gesture, and speech. In *Proc ICMI 10*. Beijing, China.

[2] Kanda, T., Shiomi, M., Miyashita, Z., Ishiguro, H., & Hagita, N. (2009). An affective guide robot in a shopping mall. In *Proceedings of HRI* (pp. 173-180).

[3] Al Moubayed, S., Beskow, J., Skantze, G., & Granström, B. (2012). Furhat: A Back-projected Human-like Robot Head for Multiparty Human-Machine Interaction. In Esposito, A., Esposito, A., Vinciarelli, A., Hoffmann, R., & C. Müller, V. (Eds.), *Cognitive Behavioural Systems. Lecture Notes in Computer Science*. Springer.

[4] Gustafson, J. (2002). *Developing multimodal spoken dialogue systems. Empirical studies of spoken human-computer interaction*. Doctoral dissertation, KTH, Department of Speech, Music and Hearing, KTH, Stockholm.

[5] Kopp, S., Gesellensetter, L., Krämer, N., & Wachsmuth, I. (2005). A conversational agent as museum guide - design and evaluation of a real-world application. In *Proceedings of IVA 2005, International Working Conference on Intelligent Virtual Agents*. Berlin: Springer-Verlag.

[6] Swartout, W., Traum, D., Artstein, R., Noren, D., Debevec, P., Bronnenkant, K., Williams, J., Leuski, A., Narayanan, S., Piepol, D., Lane, C., Morie, J., Aggarwal, P., Liewer, M., Chiang, J-Y., Gerten, J., Chu, S., & White, K. (2010). Ada and Grace: Toward Realistic and Engaging Virtual Museum Guides. In *10th International Conference on Intelligent Virtual Agents (IVA)*.

[7] Al Moubayed, S., Edlund, J., & Beskow, J. (2012). Taming Mona Lisa: communicating gaze faithfully in 2D and 3D facial projections. *ACM Transactions on Interactive Intelligent Systems, 1*(2), 25.

[8] Al Moubayed, S., & Skantze, G. (2011). Turn-taking Control Using Gaze in Multiparty Human-Computer Dialogue: Effects of 2D and 3D Displays. In *Proceedings of AVSP*. Florence, Italy.

[9] Skantze, G., Al Moubayed, S., Gustafson, J., Beskow, J., & Granström, B. (in press). Furhat at Robotville: A Robot Head Harvesting the Thoughts of the Public through Multi-party Dialogue. To be published in *Proceedings of IVA-RCVA*. Santa Cruz, CA.

[10] Al Moubayed, S., Beskow, J., Granström, B., Gustafson, J., Mirning, N., Skantze, G., & Tscheligi, M. (2012). Furhat goes to Robotville: a large-scale multiparty human-robot interaction data collection in a public space. In *Proc of LREC Workshop on Multimodal Corpora*. Istanbul, Turkey.

[11] Kueblbeck, C., & Ernst, A. (2006). Face detection and tracking in video sequences using the modified census transformation. *Journal on Image and Vision Computing, 24*(6), 564-572.

[12] Skantze, G., & Edlund, J. (2004). Robust interpretation in the Higgins spoken dialogue system. In *Proceedings of ISCA Tutorial and Research Workshop (ITRW) on Robustness Issues in Conversational Interaction*. Norwich, UK.

[13] Kopp, S., Krenn, B., Marsella, S., Marshall, A., Pelachaud, C., Pirker, H., Thórisson, K., & Vilhjálmsson, H. (2006). Towards a Common Framework for Multimodal Generation in ECAs: The Behavior Markup Language. In *Proceedings of IVA* (pp. 205-217). Marina del Rey.

[14] Larsson, S., & Traum, D. R. (2000). Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural Language Engineering: Special Issue on Best Practice in Spoken Language Dialogue Systems Engineering, 6*(3-4), 323-340.

[15] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming, 8*, 231-274.

[16] Kronlid, F., & Lager, T. (2007). Implementing the information-state update approach to dialogue management in a slightly extended scxml. In *Proceedings of the 11th International Workshop on the Semantics and Pragmatics of Dialogue (DECALOG)* (pp. 99-106).

[17] Raux, A., & Eskenazi, M. (2007). A multi-layer architecture for semi-synchronous event-driven dialogue management. In *ASRU 2007*. Kyoto, Japan..

[18] Lemon, O., Cavedon, L., & Kelly, B. (2003). Managing dialogue interaction: A multi-layered approach. In *Proceedings of the 4th SIGdial Workshop on Discourse and Dialogue* (pp. 168-177).

[19] Rich, C., Sidner, C., & Lesh, N. (2001). COLLAGEN: Applying Collaborative Discourse Theory to Human Computer Interaction. *Artificial Intelligence Magazine, 22*, 15-25.

[20] Bohus, D., & Rudnicky, A. (2009). The RavenClaw dialog management framework: Architecture and systems. *Computer Speech and Language, 23*(3), 332-361.

[21] Skantze, G. (2010). *Jindigo: a Java-based Framework for Incremental Dialogue Systems*. Technical Report, KTH, Stockholm, Sweden.

[22] Gruenstein, A., McGraw, I., & Badr, I. (2008). The WAMI toolkit for developing, deploying, and evaluating web-accessible multimodal interfaces. In *Proceedings of ICMI* (pp. 141-148). Chania, Crete, Greece.

[23] Rieser, V., & Lemon, O. (2012). *Reinforcement Learning for Adaptive Dialogue Systems.* Berlin: Springer-Verlag.

[14] http://www.cslu.ogi.edu/toolkit/