ATLAS: A generic software platform for speech technology based applications

Melin, H.

journal: TMH-QPSR
volume: 42
number: 1
year: 2001
pages: 029-042

http://www.speech.kth.se/qpsr
ATLAS: A generic software platform for speech technology based applications

Håkan Melin
Centre for Speech Technology (CTT), TMH, KTH, Drottning Kristinas väg 31, SE-100 44 Stockholm

Abstract

ATLAS is a Java software library that provides a framework for building multi-lingual and multi-modal applications, especially dialogue systems, on top of speech technology components. The design is based on a layered system model, where ATLAS sits as a middleware between an application-dependent layer and the speech technology components and implements much of application-independent functionality in the system. ATLAS is itself layered with interfaces to speech technology components at the bottom and self-contained dialogue components at the top. The layered design is both efficient and flexible and is suitable for a research environment. The framework also provides support for application-dependent layers through a structure of an application with sessions interacting with users through terminals. The terminal concept supports creating audio device-independent applications that run transparently in both telephone and desktop environments. Several speech technology components are available for use with the ATLAS framework, including text-to-speech, speech recognition and speaker verification systems. Four applications that use ATLAS have so far been developed within student and research projects at the Centre for Speech Technology (CTT), including a speech controlled telephone banking system (CTT-bank) and an automated entrance receptionist (PER).

1. Introduction

This paper presents an effort at the Centre for Speech Technology (CTT) at KTH to create a framework for multi-modal and multi-lingual speech technology applications. The framework is called ATLAS and is a Java software library that includes a set of application programming interfaces (APIs) for speech technology components. The aim has been to code much of application invariant, low-level functionality in ATLAS and to provide application programmers with a powerful, easy-to-use speech technology API. ATLAS thereby defines a multi-layered system architecture that encourages software reuse. The framework is intended for building demonstration systems in a research environment.

Human-machine interface design and usability issues are fundamental for the success of speech technology, and a demonstration system can be useful in studies on these topics. A demonstration system can also be useful in collecting speech data to support evaluation of for instance speech recognisers. Usability studies and speech technology evaluation were also the two main goals of the CTT-bank project (Ihse, 2000; Melin et al., 2001), one of the projects where ATLAS has been used.

With the growing commercial interest in speech technology based applications, and an increasing demand on research labs to do industry relevant research, it is also becoming more and more valuable to show practical examples of research advances. This often means live demonstrations of the technology in useful applications. Demonstration systems typically include several speech technology components, such as speech generation, text-to-speech synthesis, speech recognition, speech understanding, speaker recognition, and dialogue management. The components require complex interaction with each other and with audio devices, and the components are themselves complex. As a result, a demonstration system is often a complex system. To prevent system building itself to take too much effort away from the more research oriented tasks, such as improving basic speech technology components, it is important to have an efficient framework for building demonstration systems. A framework can be defined by for instance a suitable programming language, a good system architecture and reusable software components. It is important that such a framework is flexible enough to allow researchers to test new ideas, and that it evolves with state-of-the-art in speech
technology. This requirement is challenging, because it somewhat opposes the requirement for efficiency. A framework that is efficient and easy to use when building small demonstration applications may not be flexible enough when building for example state-of-the-art conversant dialogue systems.

Several publications have reported on efforts in creating frameworks for speech technology applications. A well-known platform is Galaxy-II (Seneff et al., 1998). It was developed at MIT and has been used successfully in several applications such as the Jupiter, Voyager and Orion systems. It has also been designated as the first reference architecture for the DARPA Communicator Program and is now maintained and enhanced by MITRE (Bayer et al., 2001). Galaxy-II is a client-server architecture where all interactions between servers are mediated by a programmable hub and managed by a hub script.

Jaspi$^2$ (Turunen & Hakulinen, 2000) is an agent based architecture designed with special focus on multi-linguality and user and environment adaptivity. Sutton et al. (1998) describe the OGI CSLU Toolkit that includes several ready to use speech technology components and a Rapid Application Developer tool. Potamianos et al. (1999) review efforts in defining design principles and creating tools for building dialogue systems, including architectural issues.

Several commercial companies offer platforms for developing applications with speech technology. Nuance markets SpeechObjects (Nuance, 2000) as “a set of open, reusable components that encapsulate the best practices of voice interface design”. SpeechObjects as a component technology has been standardised within the V-Commerce alliance. It is free source and claimed to be portable between platforms, spoken languages and speech engines. Philips markets SpeechPearl and SpeechMania as speech recognition and speech understanding-centric product families. SpeechPearl includes SpeechBlocks, in concept very similar to Nuance’ SpeechObjects.

Related to the creation of generic platforms are also several standardisation activities. The World Wide Web consortium (W3C) specifies markup languages for voice dialogues (VoiceXML), speech recognition grammars, speech synthesis markup, reusable dialogue components, etc. ECTF defines standards for interoperability in the Computer Telephony (CT) industry.

The ATLAS framework, presented in detail in this paper, has so far been used in four projects at CTT. It was developed within the PER (Pakucs & Melin, 2001) and CTT-bank projects (Ihse, 2000; Melin et al. 2001). Demonstration systems created within these two projects, an automated entrance receptionist and a speech controlled telephone banking system, take advantage of most features in ATLAS. The platform has also been used within the Picasso Impostor Trainer project (Elenius, 2001) and the Hörstöd project (Johansson, to appear), where subsets of its features have been used.

2. The system model

ATLAS has been designed with the layered system model shown in Figure 1 in mind. The model has an application-dependent layer on top, a resource layer in the bottom, and an application-independent layer, the middleware, in between. The upper side of the middleware is a powerful speech technology application programming interface (API), and the lower side (as seen from above) is a collection of APIs to speech technology components in the resource layer.

The middleware is itself layered. Each layer adds more powerful functionality and abstraction to the set of primitives that are offered in the speech technology API. For retained flexibility, the lower layers are always made available to the application through the API.

ATLAS is first of all an implementation of the middleware illustrated in Figure 1, but it also contains foundation classes for the application layer.

2.1. Terminology and notation

When describing software structures in the following sections, we borrow terms from the object-oriented programming paradigm as used with the Java programming language. In this terminology, a class is a collection of data and methods that operate on that data. A class is usually created to specify the contents and capabilities of some kind of object. An object created from its class specification is called an instance, or simply an object. A method is the object-oriented term for what is sometimes

\[ http://www.ectf.org/ \]

\[ http://www.speech.philips.com/ \]

\[ http://www.w3.org/voice/ \]

\[ http://www.fofoca.mitre.org/ \]

\[ http://cslu.cse.ogi.edu/toolkit \]

\[ http://www.v-commerce.com/ \]

\[ http://www.cs.uta.fi/hci/SUI/Jaspis/ \]
called a procedure or a function. For example, a circle may be defined by a radius, a location, and a color. What we would like to do with a circle is perhaps to draw it, move it and calculate its area. With an object-oriented programming language we can then define the class Circle with attributes (data) radius, location and color, and methods draw, move and getArea. Once we have the class Circle we can create instances of it, i.e. create circle objects. Each circle object has its own radius, location and color, and can be drawn or moved individually.

In this paper the word interface is used both in its general sense (for example: a human-machine interface, an application programming interface (API)) and in the object-orientation sense. In the latter case, an interface is a collection of methods and usually represent a certain aspect shared between classes. A class often implements several interfaces. In our example, the Circle class would perhaps implement interfaces Drawable, containing the method draw, and Movable, containing the method move.

New concepts, especially method names, are set in italics when introduced in the text.

3. The middleware

In this section we exemplify the contents of the various layers of the middleware as implemented in ATLAS and illustrated in Figure 1. We start at the top with the dialogue components layer and proceed towards the component APIs.

3.1. Dialogue components

A dialogue component is meant to be a powerful object that can solve a specific task within a dialogue with the user. The task can be to make a secure user login, to get the name of an existing bank account from the user, or to ask for a money amount. To solve such a task, a dialogue component must have some task-specific domain knowledge, such as knowing which customers exist and what accounts they have. The domain knowledge is often supported by an external database. Dialogue components should also be able to detect and recover from errors. An error may be an invalid response from the user such as the name of a non-existing account. If the user gives no response at all, or if he asks for help, the dialogue component should be able to provide useful help. As part of error
recovery, the dialogue component may repeat or re-formulate a previously asked question.

The purpose of the dialogue component layer is to allow a dialogue engine or the application programmer to delegate a well-defined task to an existing component, and allow the re-use of components within and between applications. If no suitable component exists for a given task, the programmer may modify an existing component, create a new one, or choose to solve the problem in some other way. In creating modified or new dialogue components, the programmer has access to all the layers in ATLAS. Dialogue components are in concept very similar to Nuance’ SpeechObjects (Nuance, 2000) and Philips’ SpeechBlocks. They also seem to be similar to dialogue agents in Jaspis (Turunen & Hakulinen, 2000).

ATLAS itself currently contains only two types of dialogue components: login procedures and enrolment procedures. The task of a login procedure is to find out who the user claims to be, and then make sure the claim is valid. A login procedure is built from a set of login operations, each of which implements a part of the login procedure. The login procedure used in a normal CTT-bank session, for example, contains two login operations. The first is an identification operation that asks the user for his name and ID-number and then looks for a matching customer identity in a database. The second is a verification operation that prompts the user to utter a randomised password and checks the answer for the correct text and for the voice characteristics associated with the claimed identity. The login procedure used in the registration call to CTT-bank, on the other hand, contains a single login operation that performs both the identification and the verification function. This operation asks the user for a unique digit sequence issued to him when he was asked to make the registration call.

While login operations implement the details of login, the login procedure itself adds procedural aspects, such as giving the user a certain number of attempts at a given operation. It also provides a single API to the dialogue engine or the application. An important point here is that it is easy for the application programmer to exchange one login procedure for another: it is just a matter of selecting another object for the task.

The task of an enrolment procedure is to elicit speech from a customer, build a representation of the customer’s speech, and store the representation in a database. In a CTT-bank registration call, a login procedure is first used to establish the caller’s identity as a valid customer. An enrolment procedure is then used to collect ten utterances from the customer. The procedure checks that each utterance is spoken correctly and asks for a repetition if needed. When ten valid utterances have been collected, the procedure trains a speaker model for the customer’s voice and stores it in a database. The same enrolment procedure is re-used in the PER demonstration system, only modified to exploit a graphical display for showing the user what utterances to speak.

Within the CTT-bank application, another set of dialogue components has been developed. They all derive from the same component called “complex question”, and their respective task is to get a money amount, to get the name of a valid account, and to get the answer to a yes/no-question. These components were created inside the application since they were not available in ATLAS, but they are candidates for being moved into ATLAS to make them easily accessible from other ATLAS-based applications.

### 3.2. High-level primitives

The high-level primitives layer currently contains an ask method and a simplified ask method. Both methods present an optional prompt from a given prompt text and record and process the answer using a set of audio processors (defined in the next section). They normally depend on methods in the services-layer for their implementation such as say and listen (also defined and described in the next section). The simplified ask method returns the top-scoring text hypothesis for the spoken answer, while the ordinary ask method gives access to the results of all participating audio processors including multiple text hypotheses and speaker information.

### 3.3. Services

The services layer provides speech and media input and output capabilities through play, say and listen methods, plus specialised retrieval methods for speech technology components (resources) of pre-defined types.

#### 3.3.1. Speech and media output

The play method loads media data from file, sends it to one or more media devices, and makes the media devices render it. The say method takes a text argument and sends it to a text-to-speech (TTS) component to generate a media stream. It then sends the generated media stream to one or more media devices like the play method. Note that both the play and the say
methods can handle multi-modal media output devices, such as speech with face animation. In this case the generated media stream contains two channels, an audio channel and a channel with parameter data for face animation.

3.3.2. Speech and media input

The *listen* method is more complex than the play and say methods. Its task is to record a segment of audio from a media device and process it. The processing is done by an optional speech recogniser and zero or more *audio processors*. An audio processor is a speech recogniser, speaker verifier, or any other object that inputs audio and outputs a result. The configuration of speech detector and audio processors to be used by the listen method is defined by a *listener profile*, central to the design of the speech input mechanism. The listener profile can specify dependencies between audio processors, such that one processor may wait for the output of another processor and use it as input to its own processing. For example, a speaker verifier A may need the output of a particular speech recogniser B to segment an utterance and another speech recogniser C for deriving an identity claim (in the case when a single utterance is used both for identification and verification of an identity). A’s dependency on B and C is then specified in the listener profile as A(B,C). In addition to audio processors given by the listener profile, the recorded audio segment can be saved to a file.

The listen method is supported by three other methods: A preparatory method sets up media streams and prepares audio processors for a new utterance according to a listener profile. A call to the preparatory method is followed by a call to the listen method itself, that triggers the start of the actual recording (the “listening”). A group of methods can then be used to retrieve results from one or more of the audio processors. When asked for results from multiple audio processors, these methods do some data fusion. Result retrieval methods normally block until results from all audio processors are available. A *maximum processing time* can be specified, however. After this time has elapsed, a method will return with the results available at the time. When all the results have been retrieved, a clean-up method should be called to release resources allocated for the listen operation.

3.3.3. Resource retrieval

Specialised retrieval methods are provided for speech technology components (resources) of each pre-defined type. Pre-defined types are currently *speech recognition engine*, *speaker verification engine*, *speech detector*, *text-to-speech engine*, *sound coder*, *media stream player*, *media stream recorder*, *graphical display*, *SQL database connection*, and *file-oriented database*. Additional and more specialised types of media stream players and recorders have also been defined, including *telephony device*, *desktop audio device*, and *audio-visual agent*. Each resource retrieval method comes in two versions: one to retrieve the default resource of a given type and one to retrieve a named resource.

3.4. Component interaction

The component interaction layer contains resource handling, media stream connections, and several structures for representing various types of information.

In resource handling, all components attached to ATLAS via a component API are abstracted to a resource, and are collected in a resource bundle. The life of a resource starts when it is created and ends when it is closed. While alive, its operation may be monitored to detect if the functionality is lost (the resource is down). Whenever the application or an object within ATLAS needs access to an attached component, it retrieves a handle to the component’s API through the component’s resource interface. This layer handles all resource types in the same way, while the services layer provides specialised retrieval methods for each resource type.

A media stream consists of one or more TCP/IP-based media channels. The end-point of a channel is a TCP socket. By convention, the media producer connects to a server socket opened by the media consumer. When the connection has been established, the producer starts transmitting data in a format specified by the consumer. In most cases, the media stream has a single channel containing audio data. The only current example of a multi-channel stream in ATLAS is the stream from a text-to-speech synthesiser to an audio-visual agent, where a second channel contains parameter data for the face animation. Media streams are created on a per-utterance basis.

Several types of information are passed between components, the ATLAS layers, and the application. The component interaction layer provides data structures to hold such information. An example is the *utterance information* structure that holds information about the contents of a spoken utterance. This may be the output of a speech recogniser and may be used as input by the application itself or
by another audio processor, such as a speaker verifier or a parser. Currently the utterance information structure supports scored text hypotheses, word timing information, and speaker information, but could be extended to support for instance syntactic and semantic information.

3.5. Component APIs

A component API has been defined for each of the pre-defined resource types listed in section 3.3. Some of the APIs are complex in that they are represented by several interfaces. The speech recogniser API, for instance, consists of a recogniser factory, a recogniser engine and a recogniser utterance. They are related in such a way that a factory creates engines, and engines process utterances (segments of audio data). Furthermore, the recogniser utterance interface uses the utterance information structures defined in the component interaction layer to represent its recognition results. The recogniser engine interface also extends the audio processor interface described in section 3.3. Similarly, the speaker verification API includes a verifier engine and a verification utterance. These are based on the SVAPI8 standard speaker verification API. Besides the functionality covered by SVAPI, the speaker verification API in ATLAS has been extended to handle ATLAS-type media streams, and to have the verifier engine extend the audio processor interface.

The TTS API also contains a factory interface and an engine interface. Utterances are handled with a method call in the TTS engine, rather than with a dedicated utterance object. The synthesis method and language are specified when a TTS engine is created and cannot be changed later. Voice properties for the selected synthesis method, such as pitch level, can be changed, however. An application can change voice or language by creating multiple TTS engines and switch between them.

4. The resource layer

As already mentioned, the resource layer refers to a collection of (speech technology) components used by an application. In this chapter we first elaborate on how components can be connected to ATLAS, and then list what components are currently available. Let us emphasise that the components themselves are not part of ATLAS, and that ATLAS is rather useless without a set of good components.

4.1. Component implementation

A component API, as the lower side of ATLAS, specifies how an application or an ATLAS layer can interact with a component, while leaving a lot of freedom for how the component is

---

8 SVAPI is a result of collaborative efforts of many companies, including Novell, Dialogic, IBM, Motorola and many others.
actually implemented. Since ATLAS is implemented in Java and the component APIs are defined in terms of Java APIs, the component as such must be a Java object. But what if we already have a speech recogniser engine written in, for instance, C++? Then we can create a pseudo-implementation of the engine in Java that uses the existing C++ program to do the actual job. Figure 2 illustrates four examples of how a speech recognition engine (labelled ASR in the figure) can be connected to ATLAS through the component API.

In the first example, the engine already has a Java implementation of the component API. Either the engine is coded in Java or it is coded in a native language (C/C++) but has a Java wrapper using the Java Native Interface, JNI.

In the second example, the engine supports another API than ATLAS’ component API. This may be an industry standard API, such as Microsoft’s SAP® or Sun’s JSAPI®, or an engine vendor-specific API (for example Philips® API). Provided the ATLAS API can be mapped to the other API, a pseudo-implementation of the ATLAS API could be created that operates as a bridge between the two APIs. Such a bridge can possibly be used with other engines that support the same standard API.

In the previous two examples, the engine is likely to execute in the same process as ATLAS itself, while in the remaining two examples the engine may be implemented as a server in a separate process. The third example illustrates a plain server implementation, where a small pseudo-implementation of the ATLAS API communicates with the server through some inter-process communication mechanism, such as the Common Object Request Broker Architecture® (CORBA), Java Remote Method Invocation® (RMI), or the CTT Broker®.

The fourth and final example in Figure 2 indicates the possibility to interface to an engine that is integrated into another speech technology system, such as the DARPA Communicator®. This could include interfacing several other Communicator engines (text-to-speech engine, parser, etc.) at the same time through a single bridging mechanism. Alternatively, each single engine could be attached directly, like in examples two and three.

4.2. Available components

In this section we list the currently available components that implement an ATLAS component API and thus can be used with ATLAS. We pay special attention to how each component is connected to ATLAS and give references for the underlying technology, but otherwise keep descriptions very brief. More detailed descriptions for some of the components can be found in Melin et al. (2001). Three components are available as internal resources executing in the same virtual machine as ATLAS (Figure 2, example 1): an energy and zero-crossing rate based speech detector, a sound coder and a file-oriented database. (The two latter components are also available as CTT Broker servers, see below.)

Two components use an industry standard API (JDBC that has been chosen to be the component API for SQL databases in ATLAS) to access an SQL database (Figure 2, example 2): one interfaces to a MySQL® database and the other to a Borland InterBase® database. Note that these ATLAS components add very little to the JDBC driver itself; it merely defines the name of the driver and loads the driver into the virtual machine.

The remaining components are implemented as clients to CTT Broker servers, (Figure 2, example 3), including:

- a text-to-speech component using RULSYS (Carlson et al., 1982) for text-to-phone conversion plus GLOVE (Carlson et al., 1990) or MBROLA (Dutoit et al., 1996) synthesisers. Several Swedish and English voices are currently available, including Lukas (Filipson & Bruce, 1997) and the Infovox® voices Ingmar, Annmarie and Roger. It can generate media streams for multi-modal output (face and voice) (Beskow, 1995).
- a StarLite speech recogniser (Ström, 1996). Acoustic triphone models trained on SpeechDat databases (Höge et al., 1997; Elenius, 2000) are available for Swedish and English (Salvi, 1998; Lindberg et al., 2000).
- a speaker verifier based on GIVES. Text-dependent modes for Swedish and English are available (Melin & Lindberg, 1999; Melin, to appear).

---

9 http://www.microsoft.com/speech/
10 http://java.sun.com/products/java-media/speech/
11 http://www.speech.philips.com/telephony
12 http://www.corba.org/
13 http://www.javasoft.com/products/jdk/rmi/
14 http://www.speech.kth.se/broker/
15 http://fofoca.mitre.org/
16 http://www.mysql.com/
17 http://www.borland.com/interbase/
18 http://www.infovox.se/
• a media device with animated agent output and audio-only input (simplex mode) (Beskow, 1995; Gustafson et al., 1999).
• a media device, “digitiser”, with desktop (simplex) audio output and input based on the Snack toolkit (Sjölander & Beskow, 2000).
• an ISDN media device with telephony call handling and (simplex) audio output and input.
• a sound coder component. Performs audio format conversion, speech parameterisation for the speech recogniser and can fork audio streams. Also available as an internal component.
• a file-oriented database. Also available as an internal component.

In addition to the above CTT Broker-based components, a registry component interfaces a registry in the Broker that keeps track of host-specific servers. The text-to-speech, speech recognition, agent, digitizer, sound coder and file database Broker servers were originally developed as part of other projects at CTT. They have recently been improved and adapted to work well with ATLAS.

4.3. The CTT Broker

The CTT Broker (Lewin, 1997) is an architecture for inter-process communication that is helpful when building modular and distributed systems. It was initially developed within the ENABL (Bickley & Hunnicut, 2000) and August (Gustafson et al., 1999) projects. The Broker is used by ATLAS to communicate with several speech technology components implemented as Broker servers, as indicated above. It is currently also used in the AdApt dialogue system (Gustafson et al., 2000), where, on the other hand, ATLAS itself is not used.

The primary function of the Broker is to pass message strings between servers through TCP ports. To manage this, it also keeps track of what servers are connected. The basic, lightweight protocol uses a short header for the Broker’s own use attached to the actual message string. The header includes a message type indicator and address information, where message types exist to connect and disconnect a server, to send procedure or function calls to a server, and to send a return or error value in response to a function call. It is up to each server to define syntax and semantics for the actual message strings – the Broker simply passes this string from sender to receiver without interpreting or altering it. The string based message protocol and the use of TCP port based connections make the operation of the Broker platform independent, in that servers can run in any programming environment and operating system that supports TCP connections. The Broker itself is implemented in Java and can therefore run on any platform that supports Java.

A secondary function of the Broker is to start servers on demand, and to detect when a server is closed. It uses a database of startable servers that defines what servers can be started and how they are started.

To aid the creation of servers, software libraries have been created for several programming languages, including Java, C, C++, Tcl, Perl and Prolog. With these libraries a server can register itself with the Broker and make calls to a remote server using constructs in the used programming language, rather than handling low-level TCP connections directly. For example, with the Java library a server creates an instance of the BrokerClient class and calls the instance’s connect method. It can then make remote calls to another server by calling a method callFunc, giving the name of the remote server and the message string as arguments. The callFunc method blocks until the Broker sends a reply and then either returns a value or throws an exception.

In addition to the basic call functionality, some of the libraries (currently Java, C++ and Tcl) provide a parser for the contents of a message string that can route calls to classes, instances and attributes inside a server. Language constructs are also available to represent classes and instances in the server. With this mechanism, the concept of remote objects is supported. The remote object concept, the parser, and the corresponding message structure are entirely optional, but are used by all servers currently available through the ATLAS platform.

Using the remote objects concept, an event mechanism has been implemented using a publication metaphor. A server creates a publication for publishing certain information, and servers subscribing to the publication gets an update message every time new information is published. This event mechanism is for instance used by the Broker itself to make server connection status information available. By subscribing to such a publication, an application can for instance know when a server is lost. ATLAS uses this feature with all Broker-server based resources. When a Broker-server based resource is created, ATLAS automatically subscribes to status information for the
corresponding server connection. ATLAS is then notified if the server is lost, and can take measures to for instance re-create the resource. The event mechanism is currently available in the Java and C++ libraries only.

The CTT Broker architecture has similarities with other inter-process communication architectures\(^\text{19}\). The Galaxy-II hub, for instance, also organises servers in a star topology where all server-to-server messages pass through the hub. The hub has a programmable controller function, however, that the CTT Broker has not.

CORBA and Java RMI provides support for manipulating remote objects almost as if they were local objects. Similar functionality can be achieved with the Broker and its support libraries. It is left to the Broker server designer, however, to provide client-side APIs that allow remote objects to be manipulated as if they were local objects. Such client-side APIs (stubs) are generated automatically with CORBA and Java RMI.

Audio and other binary streams are usually not sent through the CTT Broker. Instead, servers communicate through the Broker to setup direct connections where binary data is transmitted. This is the same as in Galaxy-II.

5. Applications

5.1. Application support

Apart from providing an implementation of the middleware illustrated in Figure 1, ATLAS provides a set of support classes for the application-dependent layer of a system. This includes interfaces and super classes for application, session and terminal classes.

The provided super classes can be used to create applications with the structure illustrated in Figure 3. The idea is that the application corresponds to an object that is created once. The application can then create session objects whose lives correspond to physical sessions of interaction with a user through a terminal. It is usually the session object that does something interesting using the speech technology API in ATLAS. The current implementation limits the number of concurrent sessions to one, for reasons of simplicity. We believe this to be sufficient for most research situations.

Each session object is connected to a single terminal. A terminal may be telephony based, in which case the session naturally corresponds to a telephone call, or desktop based. With a desktop-based terminal the session metaphor does not come as naturally as with a telephony-based terminal. It is left to the implementation of the terminal object to decide when a session should start, and to the terminal or the session to

\(^{19}\) URL references for CORBA, Java RMI and Galaxy-II were given in section 4.1.
decide when a session should be terminated. Common to all terminals is that they provide a means for initiating a session with the application, and that they are associated with an audio output and an audio input device. Optionally, a terminal may also have a means to close a session and may be associated with a display.

One of the key features of ATLAS and the arrangement illustrated in Figure 3 is that applications and sessions can interact with any type of terminal transparently (as long as they do not require particular properties of specific terminals). In CTT-bank, for example, a session normally interacts with the user via a telephony-based terminal, but it can also use a desktop-based terminal. In fact, this was often exploited during the development phase of the project. The desktop terminal could even be extended with output through an audio-visual agent. To take full advantage of multi-modal output, however, the session code needs to be modified to send requests for animated gestures to the agent, to make the agent look more alive. With such modifications the session would still run with a telephony-based terminal, since gesture requests are simply discarded if the terminal cannot visualise them.

Besides sending audio through the audio output device associated with the terminal connected to a given session, the application or the session may choose to add other output devices. This is exploited in PER, where system output is always sent to the audio-visual agent sitting by the gate, even if the current session interacts with a user through telephone. This is to indicate to a newly arrived person at the gate that the system is currently busy talking to somebody else.

ATLAS has been internationalised with respect to the language spoken within the application. That is, assuming the application dependent part of an application is also internationalised, the application can be localised to a new language. Localisation in this case involves translating text elements related to generating system prompts and interpreting user responses, and adding resources for the new language (or making sure the existing ones support the new language). All such text elements within ATLAS, i.e. in its

5.2. Examples

In this section we shortly describe four systems that use ATLAS, as examples of how the platform can be used. For each system, we explain its task, what has been coded in its application-dependent layer, which ATLAS layers are used, and what speech technology components are included in the resource layer.

All four systems have in common that their application-dependent layer is coded in Java and that it uses ATLAS application support classes to implement application and session classes.

5.2.1. CTT-bank

CTT-bank is a speech controlled telephone banking system (Ihse, 2000; Melin et al., 2001). Customers identify themselves to the system by

---

20 Internationalisation is the process of designing an application so that it can be adapted to various languages and regions without engineering changes.

21 Localisation is the process of adapting software for a specific region or language by adding locale-specific components and translating text.
saying their name and a short digit sequence. The digit sequence is chosen by the customer himself during registration, and is used to make the identification phrase unique. After claiming an identity, he verifies the claim by repeating a four-digit password generated by the system. Once allowed access to the system, the user can check account balance, list recent transactions, and transfer funds between accounts.

The application-dependent layer defines several dialogue components to implement the banking services and part of the registration dialogue. Dialogue components use methods and objects in various ATLAS layers for their implementation. ATLAS dialogue components for enrolment and login are extended and specialised, and used to implement registration and user authentication dialogues. Specialisation includes using an error-correcting code with a seven-digit registration number used to authenticate the user during the registration call, and changing prompt texts to fit the application.

The resource bundle contains a speaker verification engine, several speech recognition engines, several text-to-speech engines, a speech detector, two ISDN terminals, a desktop-based terminal, a sound coder, a file-oriented database and a MySQL database driver. The multitude of speech recognition engines is needed because the used speech recogniser does not support online grammar modification. One engine is therefore created for each specialised grammar used in the application (Melin et al., 2001).

5.2.2. PER

The PER system (from Prototype Entrance Receptionist) is installed at the central entrance of the Department of Speech, Music and Hearing. The system in its current state is basically a voice-operated lock: employees at the department may open the door by saying their name followed by a random digit sequence displayed on a screen. Speech recognition and speaker verification is used to authenticate the user, and an animated agent gives the user feedback by greeting him or asking him to try again. The physical installation includes a screen, a high-quality microphone, a relay to unlock the door, and several sensor devices to detect the presence of a person. In the future, the system will be extended to engage in dialogue with visitors to provide assistance.

The system was initially developed as part of a student project (Armerén, 1999). It has recently been re-designed to employ the ATLAS platform, and several improvements have been made (Pakucs & Melin, to appear).

Most of the current dialogue is implemented by ATLAS dialogue components for enrolment and login. For future development, a general-purpose dialogue manager, SESAME, will be added on top of ATLAS. The current application has been localised to Swedish and English. It is not a prioritised task, however, to keep system extensions, such as more advanced language understanding and dialogue control, bilingual.

The resource bundle contains one speaker verification engine, two speech recognition engines, and several text-to-speech engines per supported language. It also contains an animated agent, a graphical display, a speech detector, two ISDN terminals, one terminal object per detector, a sound coder, a file-oriented database and a MySQL database driver. Several of the resources are created especially for this application, including the display that presents the random password on the screen, and detectors with drivers to sensor devices.

Regarding the session metaphor used in ATLAS application support classes, each terminal object (except the telephone-based) uses a detector to decide when to trigger the start of a new session in the application. It is then up to the session logic to decide when the session has finished, i.e. when a user has left.

5.2.3. Picasso Impostor Trainer

The Picasso Impostor Trainer system was developed for a study on speakers’ ability to imitate other speakers (Elenius, 2001). A subject calls the system while sitting in front of a computer. He speaks a list of digit sequences to provide a sample of his normal voice, and the system compares the voice to a list of speaker models and selects two target speakers for imitation. The subject can then interactively practice to imitate a target speaker under controlled conditions: by listening to recordings of the target speaker, by watching a display with scores from the speaker verification system for his own practice utterances tested against the target speaker’s model, and combinations of the previous two. After each training round, the subject speaks ten utterances without feedback to test if he is able to alter his voice to get “closer” to the target speaker in the sense of the speaker verification system.

The application-dependent layer uses no dialogue components. Instead, the application uses the listen method in the ATLAS services layer to input and process utterances, and the play method to play back pre-recorded samples of target speakers. It has an elaborate GUI for system feedback to the user and for mouse input.
The resource bundle contains a speaker verification engine, a speech recognition engine, a speech detector, a sound coder, a media file database, and an ISDN terminal.

5.2.4. Hörstöd
This system was developed for investigating if a hearing impaired person can be aided by transcriptions produced by a phoneme recogniser in understanding speech during a telephone conversation (Johansson, to appear).

The application-dependent layer is fairly similar in content to the Picasso Impostor Trainer. It uses a telephony terminal for audio input and a GUI for graphical output. It uses ATLAS high-level primitives layer to output and process utterances, and no dialogue components are used. The resource bundle contains the same resources as in Impostor Trainer, except that no speaker verifier or speech detector are included.

6. Discussion
The main difficulty in creating a generic application platform is to make it both efficient and flexible. For a given application, the platform should be efficient to use to minimise development costs. But it should at the same time be flexible enough to be efficient for another type of application, and to allow adaptation to new types of applications. We believe that the layered structure employed in ATLAS is powerful in this regard. By providing low-level APIs and structures, where little is assumed about the overall application structure, we allow very diverse applications to share at least the same speech technology components. In the higher, more powerful layers, we assume more and more about application structure. These layers are efficient to use with applications for which these assumptions are valid, and may simply be ignored by applications for which they are not. To develop the platform to provide powerful layers also to new and diverse applications, we can either adapt the existing layer implementations and generalise them, or we can create parallel implementations with other assumptions regarding system structure. As a development process, we suggest to first develop new applications with whatever parts of ATLAS are useful, then to analyse the application-specific code to see what is general and what is specific to the particular application, and finally to successively move the general parts into ATLAS. This is how ATLAS can evolve with research advances.

VoiceXML\textsuperscript{22,23} is emerging as a standard markup language for representing human-computer dialogues. To relate ATLAS and VoiceXML to each other, we first try to describe the latter in the context of the system model illustrated in Figure 1. The VoiceXML standard primarily defines a specification for the interface between the application-dependent layer and a voice browser. The application-dependent layer in a VoiceXML application is very "thin" and is represented by a set of XML documents. The voice browser is an application-independent engine that implements dialogues according to given VoiceXML documents. It thus includes the functionality of the middle layer and the resource layer of Figure 1 (though it may have an entirely different structure). We therefore suggest that VoiceXML corresponds to the speech-technology API in ATLAS.

While ATLAS gives the application programmer access to all its internal layers for retained flexibility, VoiceXML provides access to rather high-level functionality in the voice browser, but not to low-level details. For instance, a VoiceXML application can tell the browser to ask a multiple-choice question, but cannot manipulate the voice browser’s speech recogniser directly (except possibly through a browser vendor’s proprietary features). VoiceXML has been created based on some assumptions about system structure and capabilities, and as a standard it also imposes corresponding constraints on what applications can be created. It is therefore efficient for those kinds of applications. Because VoiceXML is a standard, a VoiceXML application can also be executed in a standard-compliant voice browser from any vendor\textsuperscript{24}.

Creating media streams on a per-utterance basis allows demands for real-time performance in some parts of the system to be reduced, compared to if a continuous (unbuffered) media stream on a central bus were used. This is an advantage in a research system since it allows, for instance, an experimental speech recogniser to take the time it needs to prepare for and process an utterance. A slow component need not risk that other components involved in the processing of the same utterance looses any samples. We also believe per-utterance streams make system programming somewhat easier. It has a couple of disadvantages, however. Setting

\textsuperscript{22}http://www.w3.org/voice/
\textsuperscript{23}http://www.voicexml.org/
\textsuperscript{24}Several vendors market voice browsers that implement VoiceXML, including Tellme, Motorola, Nuance and Pipebeach.
up streams for every new dialogue turn or utterance takes more time than simply telling a device to start listening on an already connected media bus, possibly resulting in a slower system. It may also make it more difficult to use full duplex input/output streams, to implement barge-in, etc. Most standard APIs related to audio and speech also tend to assume a central media bus. Thus we consider using continuous, buffered media streams for the future.

For ATLAS component APIs we have in general not used public standard APIs (the exceptions are JDBC for SQL database connections and SVAPI for the speaker verification). This is because, first of all, for most current component implementations we use in-house technology developed before ATLAS was conceived, and we chose to design APIs that match the abilities of the current technology. Implementing a standard API, such as the Java Speech API (JSAPI), for the TTS for instance, would have resulted in overhead work at this stage. Second, the main candidate for a standard API in ATLAS would be JSAPI, and there are not (yet) many speech engines available that implement JSAPI. Furthermore, JSAPI in its current state does not integrate well with the corresponding Java Sound API and Java Telephony API that would enable us to maintain the audio device independence of ATLAS. Third, as we outlined in section 4.1, an ATLAS component API can be mapped to a standard API via a bridge to enable the use of engines with a standard API.

A corresponding division between internal and standard APIs is seen in Jaspis (Turunen & Hakulinen, 2000). In its input/output architecture, virtual devices are abstract units that represent more concrete engines. Virtual devices serve as the interface between engines (below) and agents and the communication manager (above) and partly correspond to ATLAS component APIs. Below the virtual device level in Jaspis are the client, server and engine levels, and standard APIs are employed between the server and engine levels (cf. Figure 2, example 2).

7. Conclusions

ATLAS has been presented as a framework for building demonstration applications with speech technology. So far it has proved useful for research in four CTT projects. The CTT-bank system has been used both in a usability study and for collecting data for evaluation of speech recognition and speaker verification performance. The Hörstöd system will likewise be used in a usability study and to test the performance of a phoneme recogniser. The Picasso Impostor Trainer was used to test how speakers are able to imitate other people’s voices. The PER system has been used to test speaker verification performance, and will be a platform for further exploitation of speech technology within CTT and our host department Speech, Music and Hearing (TMH).

The high-level speech technology API and the application support classes in ATLAS make application building easier compared to programming with speech technology components directly. Three of the current ATLAS applications (CTT-bank, Picasso Impostor Trainer and Hörstöd) were created as part of student projects. The platform has thus proved to be useful also for educational purposes.

The future development of the platform will include added support for natural language processing, such as message generation and parsing, and interfacing to other speech recognisers, for example ACE25 (Seward, 2000) developed at CTT. More technical developments are added logging facilities and support for recreating lost server-based resources for increased stability.

ATLAS is currently developed and used only within CTT, but we see a possibility to make it publicly available in the future.

8. Acknowledgements

This research was carried out at the Centre for Speech Technology (CTT), a competence centre at KTH, supported by VINNOVA (The Swedish Agency for Innovation Systems), KTH and participating Swedish companies and organisations.

9. References


Bickley C & Hunnicutt S (2000). ENABL – Enabler for engineering software using language and

25 http://www.speech.kth.se/ace/
Melin: ATLAS: A generic software platform for speech technology based applications


