Towards VoiceXML Compilation for Portable Embedded Applications in Ubiquitous Environments

Dirk Bühler
University of Ulm
Department of Information Technology
Ulm – Germany
dirk.buehler@e-technik.uni-ulm.de

Stefan W. Hamerich
TEMIC Speech Dialog Systems
Dialog Research
Ulm – Germany
stefan.hamerich@temic-sds.com

Abstract

In this paper we present an approach to embedding VoiceXML applications by an off-line compilation scheme. Our primary motivation is that while VoiceXML is an established standard for voice applications, the complexity and resource requirements of VoiceXML interpretation have so far limited its spread to application areas other than telephony-based services. In many contexts, such as ubiquitous computing, being able to embed voice applications into a larger framework is more important than the dynamic creation of VoiceXML documents that makes the processing particularly resource demanding. For these environments, we propose off-line VoiceXML compilation into ECMAScript as a solution for reducing the resource demands, thus making VoiceXML a viable option for the many small devices that replace the desktop in ubiquitous computing. We also describe our portable Java-based run-time platform that makes it possible to run a complete spoken language dialogue system on any computer supporting Java.

1. Introduction

Since its presentation in 1999 VoiceXML has become an established standard for human-machine spoken dialogue descriptions in the telephony environment and is used throughout many different services (see e.g. [1, 2]). Currently VoiceXML 2.1 (refer to [3]) is the latest version, discussions for extending the language and making it more flexible are ongoing.

However, VoiceXML has problems that hinder the spread of the technology in other areas than telephony services. These problems are mainly caused by the requirements of the interpreters. We therefore show a way of using VoiceXML for developing services, but compiling it to allow the usage of these compiled scripts on nearly every platform [4]. With this approach the resource requirements of the needed components for executing the compiled dialogue script may be efficiently minimised.

This paper is organised as follows: In the next section, the differences between telephony-based and embedded applications are discussed and the need for a more efficient treatment of VoiceXML is claimed. Next, we describe our approach to compiling VoiceXML that is at the core of the proposed solution. Subsequently, the run-time platform for executing the compiled application is discussed. Finally, we conclude by sketching some further work after relating our approach to other emerging standards and technologies, such as SALT and X+V.

2. Telephony-Based vs. Embedded Applications

VoiceXML has become the standard dialogue description language for dialogue applications. It is interpreted by a VoiceXML interpreter (VXI), such as OpenVXI [5] or OptimTalk [6]. To run VoiceXML applications, besides the VXI additional components are needed, which all together form a VoiceXML runtime platform. The additional components are:

- an HTTP server, which is needed to access external components like dialogue scripts, CGI scripts (e.g. as gateway to databases), etc.;
- an ECMAScript interpreter, which is required for interpreting inline scripts in VoiceXML dialogues.

In addition, most current VoiceXML applications are used as information systems providing access to (dynamic) database contents, such as timetable information systems. These backend systems normally reside on a different server, which is connected to the VoiceXML runtime platform by IP.

Due to these facts and the need for the mentioned additional components, such VoiceXML runtime platforms are normally running on servers with nearly unlimited memory resources. On these servers the applications are accessed via phone.

Regarding embedded devices used for speech dialogue systems, the amount of memory and processing power is strictly limited. For example, speech dialogue systems in cars (see e.g. [7, 8]) are embedded solutions. They are available as separate hardware boxes or integrated into infotainment systems. One point is that these systems are designed to control on-board devices (e.g. telephone, navigation system, etc.) so they are closed systems, without any need to access data dynamically. Additionally the size of the available memory is still limited. Therefore usual speech dialogue systems in cars have to be really modest in their memory usage, having only 128 KB available is a usual thing.

2.1. The Need for Embedded VoiceXML

Due to the requirements, as described above, it is clear that the network-based approach of processing VoiceXML, with its resource consuming infrastructure needed for processing the dialogue scripts is not suited for embedded systems.

However, since VoiceXML has become a standard for describing dialogue scripts and offers a very concise code, it makes sense to use VoiceXML as description language. So the only thing left to change is the runtime platform. There the idea is to minimise its resource consumption, which usually is needed for:
• Obtaining the documents using HTTP;
• Analysing (i.e. parsing) the documents;
• Processing the documents with the interpreter;
• Parsing and interpreting ECMAScript code contained in the VoiceXML documents.

As in embedded systems dynamic data retrieval and distributed applications are mostly unneeded, the HTTP server could be removed to have a smaller embedded runtime platform. When additionally the dialogue script is compiled, parsing of the script is not needed at run time either. So the execution component and the ECMAScript interpreter are left. To minimise resource requirements, these two should be combined. This could be done by compiling the VoiceXML dialogue scripts to ECMAScript.

ECMAScript (also known as ECMA standard 262 [9]), is a standardised scripting language having its origins in JavaScript from Netscape and JScript from Microsoft. The resulting standard ECMAScript has been adopted as scripting language for VoiceXML.

So compiling VoiceXML dialogues to ECMAScripts finally leads to an embedded VXI, which only consists of one component able to run the compiled script. The approach of compiling VoiceXML dialogue scripts is explained in detail in section 3.

2.2. Modelling Dialogues for Embedded Systems
Currently dialogue modelling for embedded systems is done by using programming languages like C or with special dialogue description languages, like described in [8]. Having an embedded VXI, VoiceXML could be used to describe embedded dialogue applications as well1. This would strengthen the role of VoiceXML for the future. Additionally, this would allow the use of existing tools and environments for embedded dialogue development.

3. Compiling VoiceXML to ECMAScript
This section describes our approach to compiling VoiceXML. ECMAScript was chosen as the target language, because it is already contained in VoiceXML, and the specification of VoiceXML relies upon it in many ways. Essentially, compilation to ECMAScript consists of transforming the VoiceXML document into a complex ECMAScript data structure that, at run time, is interpreted by a generic form interpretation algorithm also written in ECMAScript. Therefore, at run time there is no need for processing the original VoiceXML document. Furthermore, instead of being interpreted, ECMAScript may be further compiled into other representations.

3.1. Design Goals and Scope
This section explains the goals and the scope of our compilation approach. The most important requirement is to generate a light-weight result. The goal is for the generated code to conform to the ECMAScript Compact Profile, a subset of the ECMA 262 standard that avoids some of the more computationally costly features. In particular, the use of the eval() function is to be avoided in the generated code, since this functionality requires a complete ECMAScript parser and interpreter. The with statement is still used, but its use is limited to a restricted number of special objects like (application, document, and dialog). However, Compact Profile conformance may be compromised if scripts included in the document do not conform.

The compilation itself should be a light-weight process. In particular, we want to copy the ECMAScript code embedded in the source VoiceXML document without major analysis into the resulting ECMAScript document. Although beyond the scope of this paper, a light-weight on-the-fly compilation procedure could also be useful in a traditional VoiceXML interpreter.

As for the coverage of VoiceXML, for our compilation process we assume an offline operation, i.e. compilation is done strictly prior to execution. Consequently, we do not handle dynamically generated documents as produced by server-side scripts as a result of a VoiceXML <submit> transition. Another restriction applies to other types of transitions. We assume it is sufficient to consider intra-application transitions, since the VoiceXML concept of application seems the right scope for speech applications in an embedded environment. We have limited our attention to a subset of VoiceXML which is sufficient for interesting form-based mixed-initiative dialogues, ignoring elements related to transfers, for instance. The most important supported elements are mixed-initiative forms, gotos, sub-dialogs, links, and scripting.

3.2. Compilation Procedure
In the following we give an overview of the compilation procedure that is used to translate VoiceXML documents into executable ECMAScript code.

1. First, the documents are fetched from a file or web server. This might involve recursive fetches, since document transitions are allowed, but have to be known at compilation time. Separate ECMAScript code files referred to by the VoiceXML document are retrieved as well. Grammars included by the VoiceXML documents those imported within other grammars are obtained as well.
2. Next, a single data structure similar to an abstract syntax tree is constructed, containing the complete application. The main levels of representation are document, form, and form item. In this data structure, the grammar contents are included.
3. Finally, this data structure is transformed into ECMAScript code. Grammars are compiled into finite state networks and included as data structures in the generated ECMAScript code.

The resulting ECMAScript code for a <form> consists of the declaration of a dialog data structure that is later used by a generic form interpretation algorithm (FIA) to execute the form (the dialog object). In our approach, this FIA is also completely written in ECMAScript. The generated dialog data structure contains information about the form’s grammars, links, variables, form items, filled elements, and the like. Form items are contained in the dialog data structure and have a similar structure. In particular, they contain function objects for VoiceXML conditions, prompts and other executable content. Grammar data structures are passed to platform calls that

---

1 However, for embedded systems, restrictions for some concepts appear.

2 A VoiceXML interpreter, written in ECMAScript as an alternative to compilation, would depend on eval() for interpreting the ECMAScript parts in the VoiceXML document.

3 We use the term “dialog” to refer to VoiceXML dialog objects (i.e. forms) in contrast to the general notion of dialogue.
at run-time return grammar handles for use in their declaring scopes. In the resulting code, scripts and other ECMAScript expressions contained in the VoiceXML document are included as textual copies, requiring only a slight processing due to scoping issues with variable and function declarations. Executable code generated for conditions, prompts, and \(<\text{filled}>\) elements, for instance, is represented by function objects that may be called multiple times during the execution of the application.

The ECMAScript event handling mechanism is used for handling events in the VoiceXML code, i.e. a \(<\text{throw}>\) translates to a ECMAScript \(<\text{throw}>\) statement. Accordingly, event handlers are implemented as try blocks. When the generic FIA detects an event thrown by an item it tries to handle it by catching - in that order - the item’s, the dialog’s and the document’s handler callback functions. Each of these functions re-throw the event if it cannot handle it.

The current version of the ECMAScript compiler consists of ca. 800 lines of code, written in the Haskell functional programming language. The compiler includes a compiler for XML grammars conforming the W3C Speech Recognition Grammar Specification (SRGS). These grammars are compiled to a finite state network that can be easily interpreted by the runtime platform’s parser component. The grammars may contain semantic tags in the form of ECMAScript code. These code fragments are represented in the grammar data structure as function objects. The run-time environment conceptually consists of three levels:

- **The compiled application:** The first level is the ECMAScript code generated specifically for a certain VoiceXML document. This code invokes generic code reused in all compiled documents.
- **The generic form interpretation algorithm (FIA):** The generic FIA makes use of a predefined number of platform objects, such as \(<\text{rec}>\) for the speech recognition and grammar handling, as well as, \(<\text{prompt}>\) for speech synthesis. In our current prototype, the generic FIA consists of about 300 lines of code, written in ECMAScript.
- **The generic adaptor code:** This part of the platform contains objects that are usually implemented natively, i.e. here, in Java.

When executing a form, the generic FIA interprets the respective form data structure that was compiled from the VoiceXML document. First, the form data structure is initialised by a function object it provides. The initialisation procedure creates form-level grammars and the form item data structures. These structures contain important dynamic administrative data that the FIA needs, e.g. repetition and event counts. When selecting a form item, the form items’ structures are examined and condition functions objects are evaluated as necessary. Another important task of the FIA is the selection of the correct prompts according to the (repetition) count of the item.

At run-time the FIA has access to two abstract adaptor objects:

- **The Recogniser interface** handles the creation and activation of grammars as well as the actual acquisition of user input.
- **The Prompter interface** is responsible for presenting textual information and audio files to the user.

The abstract adaptor objects serve as proxies to **front-end plug-ins**, that realise the actual acquisition and presentation depending on the user’s preferences when the application is executed. Four major pure Java plug-ins have been realised:

- **The Sphinx4 plug-in** is an interface to the pure Java Sphinx4 speech recognition system [10]. Sphinx4 expects grammars in Java Speech Grammar Format (JSGF) and is able dynamically create and activate grammar rules. The grammar data structures contained in the compiled VoiceXML application contain a JSGF representation derived from the SRGS source. This representation lacks semantic tags, because the recognised text is treated as if obtained from the console (see below).
- **The FreeTTS plug-in** provides a simple Text-to-Speech synthesis functionality. Speech markup is ignored at the moment.
- **The Console I/O plug-in** realises typed input through the standard input and output streams. Typed input obtained from the input stream is processed by the parsing component that interprets the compiled grammar data structures contained in the VoiceXML application and returns a ECMAScript result frame.
- **The GUI Text I/O plug-in** (cf. Fig. 1) works in the same way, but interacts with Swing widgets instead of the standard text streams. This is mainly useful for applets and Java Web Start-based deployment.

With these plug-ins it is possible to deploy a complete pure Java end-to-end speech dialogue system, either as an applet or as a Java Web Start-enabled downloadable application that is independent of a network connection at run-time. The download is quite heavy when Sphinx and FreeTTS are used, about 20 Megabytes. However, the Sphinx4 and FreeTTS components, which account for about 95% of this load, are application-independent and have to be downloaded only once and are reused for other compiled VoiceXML applications.

The run-time platform uses the Mozilla project’s RHINO JavaScript interpreter and compiler [11] for compiling the generated ECMAScript code into Java bytecodes. Compared to a VoiceXML interpreter implemented in Java, this approach eliminates the need for XML and also EcmaScript parsing at run-time, provided that the \(<\text{eval}>\) is not used within the VoiceXML document’s ECMAScript code. Using Java makes it fairly straightforward to implement platform adaptor objects as Java classes.

Through the use of ECMAScript, virtually all the Java classes are available to compiled VoiceXML applications. This makes it particularly easy to allow VoiceXML applications to interact with a graphical user interface, e.g. implemented in Swing components (see Fig. 2). In this example of a VoiceXML form a GUI window is created, stored in a form variable, and can be used as an additional output medium. For reacting on
cable in situations where the VoiceXML application consists of static documents, i.e., are not generated dynamically by a web server. This new approach opens a new perspective for VoiceXML, since it then could not only be used for describing telephone-based applications but additionally for embedded applications.

The work presented in this paper opens an interesting perspective on a number of future directions. First of all, the coverage of VoiceXML elements has to be extended. In addition, because ECMAScript may also be used in the semantic part of grammars, grammar compilation should be more tightly integrated in the compilation framework. In fact, compiled grammars could be represented as ECMAScript objects. Even a finite state parsing technique completely written in ECMAScript seems feasible. Finally, other proposed speech technology standards such as SALT and X+V could benefit from our compilation procedure. The procedure could be used, for instance, to compile the forms contained in an X+V document into regular HTML scripts.

6. References