

Using Java Speech Grammars as Cache Memory for Databases in VoiceXML Applications

S. D. Eyono Obono, M. J. Serumaga-Zake

Abstract—VoiceXML applications capture tokens input by users and match them against a specified grammar. But the Java Speech Grammar Format does not provide for a mechanism to capture tokens unmatched by a grammar. In this paper, we show how can a VoiceXML application capture tokens unmatched by a Java Speech Grammar. We also show how to dynamically update a Java Speech Grammar so that it can recognize tokens previously unmatched. We subsequently show how such captured tokens can interact with a relational database and how to transform a grammar into a cache memory for the database.

Keywords—Cache Memory, Database, Grammar, Java Servlets, VoiceXML.

1. INTRODUCTION

VoiceXML provides a standard interface between a voice and the Internet. VoiceXML lets you create audio dialogs able to record, to capture, to digitize and to recognize speech, sounds, and DTMF (Dual Tone Multi Frequencies used by the telephone for dialing) key input. End-users are able to interact with VoiceXML applications in the form a conversation in a natural language. Field values are extracted from such conversations before being processed by the backend database. Voice is captured via speech input devices such as microphones and telephones. VoiceXML is therefore an attractive tool for the development of telephony applications.

Fig.1 depicts the architecture of a VoiceXML application. When a user calls a VoiceXML application, the user's voice is converted into text by a speech recognition engine. And

that text is matched against the tokens of the available grammar.

In case of successful parsing of that text by the grammar, the corresponding tokens are extracted form that text and they are forwarded to the Web Server.

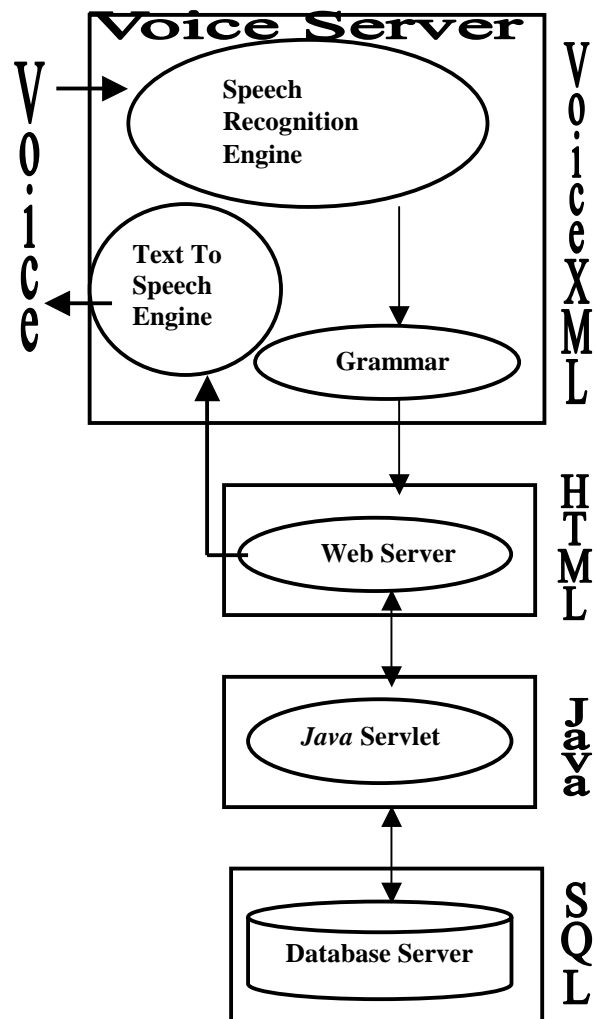


Fig.1. Architecture of a VoiceXML Application

The Web Server then forwards those parameters to a Java Servlet that interacts with a Database Server. The result expected from the Database Server is then sent back to the Java Servlet that in return forwards it to the Web Server. The Web Server finally sends back that result to the VoiceXML Application in order for it to be collected by the user. The different servers communicate with each other using parameters.

In case of failure of the parsing of that text by the grammar, the VoiceXML application has to terminate, simply because

Manuscript received June 23rd, 2003. S. D. Eyono Obono was with the Department of Information Systems, University of North West, Private Bag X2046, Mmabatho 2735, South Africa (phone: 018-389-2489; fax: 018-392-1898; e-mail: Eyono-ObonoS@uniwest.ac.za). He is now with the Durban Institute of Technology, Durban 4000, P.O Box 1334, South Africa (phone: 031-204-2707; fax: 031-204-2105; e-mail: EyonoObonoSD@dit.ac.za).

M. J. Serumaga-Zake is a BSc Honors student in the Department of Information Systems of the University of North West, Private Bag X2064, Mmabatho 2735, South Africa (phone: 083-756-9870; fax: 018-392-5775; e-mail: jzakes@hotmail.com).

This paper is part of the research activities of the Center of Excellence in Web Based Telecommunications Applications of the University of North West (South Africa). The Center is funded by three South African companies and organizations: Telkom (www.telkom.co.za), Grintek (www.grintek.co.za), and Thrip (www.nrf.ac.za/thrip).

VoiceXML does not provide a mechanism to save a text not matched by the grammar.

Why save a text that has failed the test of the grammar's parser? Before answering to that question, it is important to recall the basic elements of VoiceXML so that we can formally define the concepts to be used later on. We will then introduce the idea of using a Java speech grammar as a cache memory for the database. The implementation of that idea will lead us to propose a solution to the problem of saving a text that has failed the test of the grammar's parser.

2. BASIC CONCEPTS OF VOICEXML

The entire VoiceXML specification is huge and cannot be covered in a paper of this nature. Therefore we will only consider the subset of VoiceXML that is directly relevant to the purpose of this paper.

2.1. VoiceXML Application and Session

A VoiceXML application is a set of VoiceXML documents put together with the purpose of performing certain tasks for the end-user. The execution of a VoiceXML application by a user is called a *session*. Because VoiceXML is a markup language, all the elements of VoiceXML are defined in term of tags.

2.2. Elements of a VoiceXML document

A VoiceXML document is made of a *header* and a *body*. The document's header defines the XML version of the document as well as the document type (VoiceXML document). A VoiceXML document's body may contain the following elements: metadata, properties, user-defined variables, exception handlers, scripts, forms and menus. Metadata is introduced by the *meta* tag and consists of general information on the document such as its author, its keywords, its maintainer, etc. Properties are defined by the tag *property* and are used to set the values of the environment of the application including values for the time-out, the audio confidence level, etc. User-defined variables are introduced by the tag *var* and developers use them to represent objects of their choice. Exception handlers are defined by the tag *catch* and developers use them to define the behavior of the application when certain events occur. Scripts are small programs written in a client-side scripting language such as JavaScript and VBScript. Forms and menus are also called *dialogs*. A *sub-dialog* is dialog that is nested in another dialog (parent) with the purpose of calculating a value to be returned to the parent dialog. Menus are used by developers to allow users to select possible values of a field chosen from a predefined limited number of choices. The union of all the possible choices of a menu is a finite set of values. The tag for menus is naturally *menu*. Forms are defined with a tag *form*. Developers use forms to read values of relevant fields input by users. The union of all possible values input by users through a form is not necessary a finite set: that union is defined by a *grammar*.

2.3. Theory of Grammars and BNF Grammars

The concept of grammar forms part of the foundations of the theory of computer languages. The theory of computer

languages is well summarized by the Chomsky hierarchy shown on Figure 2.

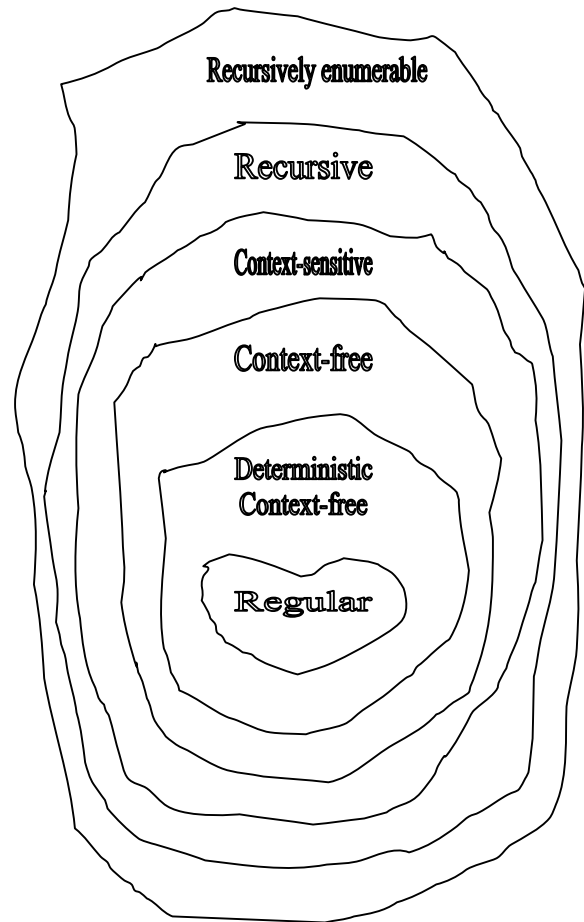


Fig. 2. The Chomsky Hierarchy of Languages

A *language* is defined from an alphabet. An *alphabet* can simply be seen as a set of symbols or letters. For example the alphabet for the English language is made of the letters from a to z, that means every English *word* is made of letters from a to z. If we restrict ourselves to the set of integer binary numbers, then the alphabet is only made of the two letters 0 and 1. The alphabet $A = \{a, b\}$ is commonly used in the theory of languages and that is in line with the fact that instructions executed by computers are ultimately expressed as a sequence of binary digits 0 and 1. Given an alphabet A , the set of all possible words built from that alphabet is called the Kleene's closure or Kleene's star of A , and is denoted by A^* . Saying that L is a language over A simply means that L is a subset of A^* .

They are different types of languages as outlined by the Chomsky hierarchy. And for each type of language, there exists a specific type of grammars corresponding to that class of languages, except for the class of recursive languages. As a result, the terms language and grammar are interchangeable. Recursively enumerable languages are defined by Type 0 grammars equivalent to Turing Machines. Context-sensitive languages are defined by Type 1 grammars equivalent to Turing Machines with bounded tapes. Context-free languages are defined by Type 2 grammars equivalent to Push Down Automata. Deterministic context-free languages are defined by LR(k) grammars equivalent to deterministic Push Down Automata. Regular languages are

defined by Type 3 grammars equivalent to Finite Automata. Needless to say that there are languages not covered by the Chomsky hierarchy. But fortunately, the type of grammars used by VoiceXML applications are covered by the Chomsky hierarchy.

The theory of computation defines a grammar as a set of production rules of the form $X \rightarrow Y$ where X and Y are words over a certain alphabet made of two disjoint subsets, the alphabet of terminals, and the alphabet of non-terminals. If for the same X we have two different production rules $X \rightarrow Y$ and $X \rightarrow Z$, then two production rules can be summarized as $X \rightarrow Y \mid Z$ where \mid stands for OR, $X \rightarrow Y$ and $X \rightarrow Z$ are then called alternatives of the production rule. Generally it is possible for X or Y to mix terminals and non-terminals. But certain restrictions on X and Y can determine the type of the grammar. For example if we restrict X to be a letter from the non-terminal alphabet with no restriction on Y , then we obtain the class of context-free languages equivalent to type 1 grammars. The presentation of a grammar as a set of such rules $X \rightarrow Y \mid Z$ is called the Backus Normal Form or Backus-Naur Form abbreviated as BNF.

Grammars used by VoiceXML are called speech recognition grammars and they are a mixture of regular grammars and context-free grammars (deterministic or not), and because the set of regular languages is included in the set of context-free languages, we can simply say that the set of speech recognition grammars is a subset of the class of context-free grammars.

2.4. Java Speech Grammars

The World Wide Web Consortium W3C recommends two types of grammars for speech recognition namely Augmented Backus Normal Form (ABNF) Grammars and XML Form (XMLF) Grammars. XML Form grammars are grammars that are expressed using the XML syntax. ABNF grammars are grammars that are expressed using the Backus-Naur Form modulus few additional features. Java Speech Grammars are simply an example of ABNF grammars. We will therefore restrict ourselves to ABNF grammars because the focus of this paper is on Java Speech Grammars. That restriction does not make our results less significant: the W3C speech recognition grammar specification ensures that any ABNF grammar can easily be transformed into an XMLF grammar and vice-versa.

The concept of ABNF grammars such as Java Speech Grammars is built from the concept of BNF grammars. It Means an ABNF grammar looks like a BNF grammar but there are few more features added to ABNF grammars such as the Kleene's star, the Kleene's plus, parentheses, weights, square brackets, etc. All those features make it possible for VoiceXML applications to process sentences expressed in natural languages. But because we are only interested in how to use grammars as cache memories for databases, we are going to even restrict further the type of ABNF or Java Speech grammars we are dealing with. We will be dealing with ABNF grammars or Java Speech grammars only made of production rules of the form $X \rightarrow X_1 \mid X_2 \mid \dots \mid X_n$ with X_i simply made of terminals or tokens for each i between 1 and n . A word will be accepted by the grammar if and only if that word belongs to the set $\{X_1, X_2, \dots, X_n\}$. It is easy to

understand that words that are accepted by the grammar will be further processed by the database. But it is less understandable to still be interested in words that have been rejected by the grammar.

3. DATABASE COMMANDS

Recall that our ultimate aim is to use the voice as an interface between a user and a backend database. So let's have a few words on databases. Database developers use SQL data modification and query commands such as INSERT, UPDATE, DELETE, and SELECT in order to allow end-users to interact with the database. We will see how words that are rejected by grammars are still useful for each of those commands. Because most of those commands use conditions, we first need to say a few words on those conditions.

3.1. CONDITIONS

Conditions are used by SQL to make comparisons between attributes values and other attributes values, or between attributes values and literal values. Comparison operators includes =, != also known as \diamond , $<$, $>$, $<=$, $>=$, between, in, etc. It is possible for a condition to be made of basic conditions linked by Boolean operators such as AND, NOT, and OR.

3.2. SELECT

Here is a simplified syntax of the SELECT command

```
SELECT attribute_1, attribute_2, ... , attribute_n
FROM table_1, table_2, ... , table_m
WHERE condition
```

The condition in the WHERE clause of the SELECT command may require that literal values are entered by the user. Those values must be checked by a grammar before being sent to the SELECT command.

3.3. INSERT

Here is a simplified syntax of the INSERT command

```
INSERT
INTO table (attribute_1, attribute_2, ... , attribute_n)
VALUES (value_1, value_2, ... , value_n)
```

The row of values to be added by the INSERT command may require the user to enter literal values. Those values must be checked by a grammar before being sent to the INSERT command.

3.4. UPDATE

Here is a simplified syntax of the UPDATE command

```
UPDATE table
SET (attribute_1=value_1, ... , attribute_n=value_n)
WHERE condition
```

The new values to be assigned by the UPDATE command may require the user to enter literal values. Those values must be checked by a grammar before being sent to the UPDATE command. The same applies to the literal values that might be used by the condition in the WHERE clause.

3.5. DELETE

Here is a simplified syntax of the DELETE command

```
DELETE  
FROM table  
WHERE condition
```

The condition in the WHERE clause of the DELETE command may require that literal values are entered by the user. Those values must be checked by a grammar before being sent to the DELETE command.

In short, SQL requires that any literal value to be entered by the user must be checked by a grammar before being sent to a database command. That leads us to the idea of using grammars as cache memory for databases.

4. GRAMMARS AS CACHE MEMORY FOR DATABASES

Cache memory is a fast but small memory used to hold the most commonly used words of the main memory. Because cache memory is far smaller than the main memory, many words of the main memory do not appear in the cache memory. Any search of a word starts from the cache. The main memory is searched only when the cache search has failed. A successful cache search is called a cache hit. A failed cache search is called a cache miss. And the chances of successfully hitting the cache are measured by a probability called the hit ratio. In case of a cache miss, the searched word is added to the cache after being found from the main memory. Memory access is considerably speed up with a good cache hit which itself depends on the effectiveness of the cache update.

Let's recall our restricted definition of a grammar as a set of productions rules of the form $X \rightarrow X_1|X_2|\dots|X_n$ with X_i simply made of terminals or tokens for each i between 1 and n . We have also said earlier on that SQL requires that any literal value to be entered by the user must be checked by a grammar before being sent to a database command. It is therefore a natural idea to consider a grammar as a cache memory for the database. Let's look at the behavior of such a cache for each database command.

4.1. CACHING FOR THE SELECT COMMAND

Literal values are input by the user and used in the WHERE clause. In case of a cache hit, which means the literal values have been successfully matched by the grammar, the SELECT command is simply carried out. In case of a cache miss, meaning a literal value is not matched by the grammar, that literal value still has to be sent to the SELECT command for execution. The literal value is then added to the grammar.

4.2. CACHING FOR THE INSERT COMMAND

Literal values are input by the user and used in the VALUES clause. In case of a cache hit, which means the literal values have been successfully matched by the grammar, the INSERT command is simply carried out. In case of a cache miss, meaning a literal value is not matched by the grammar, that literal value still has to be inserted to the database. The literal value is then added to the grammar.

4.3. CACHING FOR THE UPDATE COMMAND

Literal values are input by the user and used in the SET clause. In case of a cache hit, which means the literal values have been successfully matched by the grammar, the UPDATE command is simply carried out. In case of a cache miss, meaning a literal value is not matched by the grammar, that literal value still has to be assigned to an attribute of a relation of the database. The literal value is then added to the grammar.

4.4. CACHING FOR THE DELETE COMMAND

Literal values are input by the user and used in the WHERE clause. In case of a cache hit, which means the literal values have been successfully matched by the grammar, the DELETE command is simply carried out. In case of a cache miss, meaning a literal value is not matched by the grammar, that literal value still has to be sent to the DELETE command for execution. The literal value is then added to the grammar.

When a literal value input by a user is successfully matched by a grammar, that literal value is sent to a SQL command. But when a literal value fails the test of the grammar, VoiceXML does not provide any mechanism to store that value in a variable. And that is an obstacle to the cache update. It is therefore important to engineer a mechanism able to store in a variable a literal value that has failed the test of the grammar.

5. GRAMMARS OR CACHE UPDATES

Suppose we have a production rule of the form $X \rightarrow X_1|X_2|\dots|X_n$ with X_i simply made of terminals or tokens for each i between 1 and n . A word will be accepted by the grammar if and only if that word belongs to the set $\{X_1, X_2, \dots, X_n\}$. Ideally, we would like to have a grammar that can accept any possible word over our alphabet and then send that word to the database for further scrutiny. In theory, the Kleene's star operation solves that problem with BNF grammars. But in practice, ABNF grammars such as Java Speech Grammars do not implement the Kleene's star properly: they require a space between two consecutive tokens.

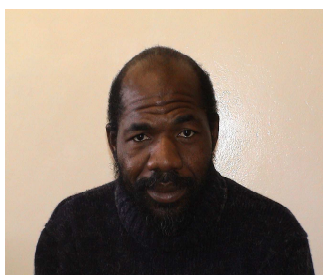
Our solution for that problem is to have a VoiceXML sub-dialog that requests the user to spell the unmatched word letter after letter. The sub-dialog then concatenates all the captured letters and sends them back to the VoiceXML application as a complete word. Because such a word was not recognized by the grammar, it must be added to the grammar because we have designed the grammar as a cache. A way of doing that is to store the grammar in a text file that can be updated not by a VoiceXML script (they cannot write in text files) but by a Java servlet.

6. CONCLUSION

Using grammars as cache memory for databases in VoiceXML applications may contribute to the speeding up of the performance of such applications. A way of speeding up the parsing process of such grammars may be to implement such cache memories directly at the hardware level.

REFERENCES

- [1] Daniel I. A. Cohen, *Introduction to Computer Theory*, John Wiley & Sons, Inc, 1986.
- [2] Andrew S. Tanenbaum, *Structured Computer Organization*, 4th ed., Prentice Hall, 1999.
- [3] J. Handy, *The Cache Memory Book*, 2nd ed., Academic Press, 1998.
- [4] P. Atzeni, S. Ceri, S. Paraboschi, R. Torlone, *Database Systems - Concepts, Languages and Architectures*, McGraw-Hill, 1999.
- [5] *Speech Recognition Grammar Specification, Version 1.1*, 26th June 2002, World Wide Web Consortium, Available: <http://www.w3.org/TR/speech-grammar>
- [6] *Voice Extensible Markup Language (VoiceXML), Version 2.0*, 28th January 2003, World Wide Web Consortium, Available: <http://www.w3.org/TR/Voicexml20>
- [7] *Information Technology --- Database Languages --- SQL*, ANSI Standard ISO/IEC 9075:1992.
- [8] *Database Languages SQL*, ANSI Standard X3.135-1992.



SD Eyono Obono was born in Yaounde (Cameroon) on October the 15th 1967. He completed his primary and secondary education in Yaounde with a “Baccalaureat C” (Mathematics and Physics)

obtained in 1986. He was then awarded a bursary by the Cameroonian government to pursue his tertiary education in France where he obtained a BSc degree in Computer Science (Nancy I) in 1990, a BSc honors degree in Computer Science (Nancy I) in 1991, a MSc degree in Computer Science (Rouen) in 1992, and a PhD in Computer Science (Rouen) in 1995.

He is currently Associate Director in the Department of Information Technology at the Durban Institute of Technology in South Africa. He was previously Associate Professor at the University of North West (South Africa) when he started working on voice applications. His previous research interests include algorithms for Np Complete problems of the free monoid.

Dr. Eyono Obono is married and is the father of three sons and one daughter.



JM Serumaga-Zake was born in Kampala (Uganda) on November the 11th 1975. This paper forms part of his research project supervised by Dr SD Eyono Obono and submitted in part

fulfillment of the requirements of the BSc Hons degree in Computer Science at the University of North West (South Africa).