HIV Testing Site Locator: Applying Current Computing Trends to Voice Applications

Mathe Maema[1] and Alfredo Terzoli[2]
Department of Computer Science
Rhodes University
Grahamstown, South Africa
E-Mail: [1]mathe@rucus.ru.ac.za [2]a.terzoli@ru.ac.za

Lorenzo Dalvit
Department of Education
Rhodes University
Grahamstown, South Africa
E-Mail: ldalvit@gmail.com

***Abstract*—Voice applications are increasingly growing in popularity as computing technologies advance. This is a positive development because voice communication offers a number of benefits over other forms of communication; even though information can only be presented serially. As a means to ameliorate this limitation, we have proposed a futuristic architecture for building voice applications deduced from current trends in computing. A functional prototype system has been built to validate this architecture. This paper focuses on the implementation details of this system. From an architectural point of view, the system has a rich back-end with a rule-based dialog driver and an ontology-driven knowledge base. Further, it has a front-end constructed using VoiceXML, the de facto standard for voice automated applications.**

***Index Terms*—Ontologies, Rules, Voice applications, VoiceXML.**

I. Introduction

IN computing, the old adage about change being a constant is certainly true. However, what is uncertain is whether the change is driving the many trends that exist or it is the trends that are driving the many changes that happen. Whichever comes first, the point is that there are many trends and/or changes in computing.

We have trends that suggest a move from imperative to declarative programming [1]. We have trends that signify a move towards broad adoption of ontologies in any system that enables knowledge and/or information exchange [2]. A popular definition for ontologies describes an ontology as "an explicit specification of a conceptualisation" [3]. As a direct consequence to the adoption of ontologies, we have trends that indicate a move away from databases to knowledge bases, where inference engines are used to support reasoning activities. There are certainly other examples that may indicate how various areas of computing, such as mobile and pervasive computing, may be influencing a shift from one trend to another; but for the sake of brevity, we will not attempt to state them.

Based on our interest in voice applications, we analysed the above-mentioned trends and many others, with the view to improving the overall experience offered by these type of applications. Consequent to our analysis, we then proceeded to propose an architecture for building voice applications (see preliminary proposal in [4]). We believe voice communication remains good for many situations where visual communication may not be appropriate. For example, voice, as opposed to visual communication, may be better for delivering services to the semi-literate users, or to users who may not be in a position to use their eyes; because they are possibly busy with another task (e.g. driving). Therefore, we hope our architecture will contribute to creation of voice applications that offer acceptable user experience.

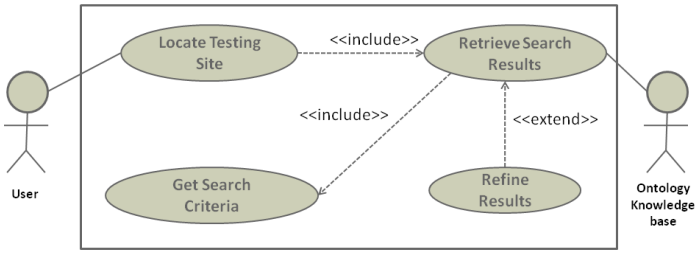

Figure 1. Use Case Diagram for HIV Testing Site Locator

For purposes of validating our architecture, we decided to build a simple voice application system. While any system would have done, we decided on one that would generate spin-off benefits for the Rhodes University community. For this reason, with the objective of reinforcing the ongoing 'know-your-status' campaign, we decided to build a system that would help individuals locate information about service providers that offer Human Immunodeficiency Virus (HIV) testing. We named our system, HIV Testing Site (HTS) Locator. In this paper, we will provide more details on how we implemented this system. Our objective is to provide a practical insight into the implementation process that may, hopefully, serve as an assurance that indeed, our architecture is functionally viable.

The rest of the paper is structured as follows. Section II presents a use case for the HTS Locator. Section III provides the architectural design of the entire system. Section IV and Section V present implementation details of the front-end and the back-end, respectively. This is followed by Section VI, which provides an insight into how testing was done. Finally, Section VII provides our concluding remarks.

It should be noted that in this paper, front-end refers only to what is rendered to the user and back-end is used loosely to describe all that happens in the background. Thus, what would otherwise be referred to as the middle tier when using architectural design patterns such as n-tier architecture, is qualified as part of the back-end. This is done to simplify the presentation of the paper and is by no means a reflection of lack of separation of concerns in the implementation.

II. The Use Case

The use case shown in Figure 1, captures the overall functionality of the HTS Locator system. As stated already,

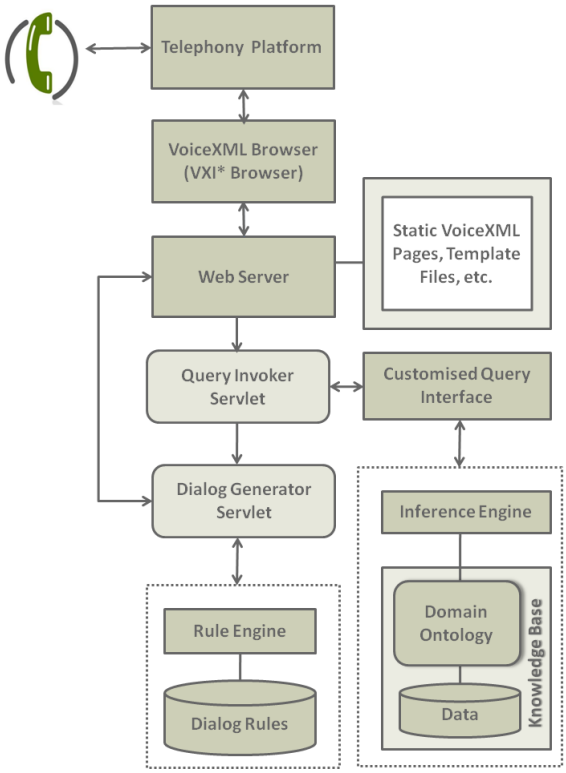

Figure 2. Prototype Implementation Architecture

the key objective of this system is to help users locate HIV testing sites aligned to their individual needs.

According to the use case, in order to locate a testing site, a search query is initiated to retrieve information from the ontology knowledge base. This query is generated by obtaining search criteria that needs to be matched. In this particular case, the criteria is determined by the cost of testing and the proximity of a site to Rhodes University campus. Once retrieved, the results may be refined. The refinement process together with the overall implementation of the system will be discussed in sections to follow.

III. System Architecture

Figure 2 provides a picture of the architecture used to implement the HTS Locator system. Based on our interpretation of current trends, we proposed this architecture as an extension to the VoiceXML architectural model, described detail in [5].

Essentially, in our proposed architecture, we have added a rich back-end. This back-end draws inspiration from the branch of artificial intelligence that deals with expert systems in that it makes use of inference engines; a knowledge base; and most importantly uses executable knowledge representation formalisms i.e. rules and ontologies. Combined, these features qualify our architecture as knowledge-driven.

IV. Front-end Implementation

With voice applications, the conversations (dialogs) that the user may have with the computer qualify as the user interface for the application. For this reason, the key to implementing the front-end of voice applications lies with creating effective conversations.

With the above in mind, we wrote out conversations for different scenarios in a drama script format to simulate communication by the user and the system. On the basis of these scripts, we created pages using VoiceXML 2 specification. An example page is shown below. Essentially, this page includes two dialogs represented within *form* tags. The first dialog greets the user and the *goto* tag moves it to the options dialog. Within the options dialog, the user is asked to make choices that would aid in locating a suitable provider. These choices are submitted to a servlet on the web server for further processing.

```
<?xml version="1.0"?>
<vxml version="2.1" xmlns="http://www.w3.org/2001/vxml"
    >
  <!--Error handling logic and variable declaration
      ommitted-->

<!-- Start initial dialog -->
  <form id="greeting">
    <block>
      <audio src="">
        <!-- Welcome message -->
      </audio>
      <goto next="#options"/>
    </block>
  </form>

 <!--Present user with options -->
  <form id = "options">
    <!--Determine cost preferance -->
    <field name ="costParam" type="boolean?y=1;n=2;">
      <prompt>Are you interested in a free or low cost
          testing site? Press 1 for yes and press 2
          for no.</prompt>
      <filled>
        <if cond="costParam=='true'">
          <assign name="cost" expr="'lowCost'"/>
        <elseif cond="costParam=='false'"/>
          <assign name="cost" expr="'moderateCost'"/>
        </if>
      </filled>
    </field>

  <!--Determine distance preferance -->

  <!--Submit user selection -->
  <block>
     <submit next="http://localhost/voiceLocApp/query"
         method="post" namelist="distance cost"/>
  </block>
</form>
</vxml>
```

In this section, we did not provide details of the generation of the pages. To an extend, this is because the task itself is not complicated. It is however worth mentioning that these pages were either generated statically or dynamically by the back-end.

V. Back-end Implementation

There are at least two important functions performed by the back-end. These are, respectively, query answering and dialog processing. In this section, we will provide high-level details of how each function was implemented.

However, before proceeding, it is worth mentioning that we used Java as the glue for making the various components of the back-end work together; as it might have be deduced from the use of servlets in Figure 2. We chose Java precisely because of trends that indicate that we are steadily moving into a generation of Java based tools within telephony. (This is evidenced by the growing adoption of platforms like Mobicents, a popular open source Voice over IP (VoIP) platform that is based on efforts by the Java community [6].)

A. Implementing Ontology-based Query Answering

As expected, the first critical step for providing ontology-based query answering was to build the ontology to be used.

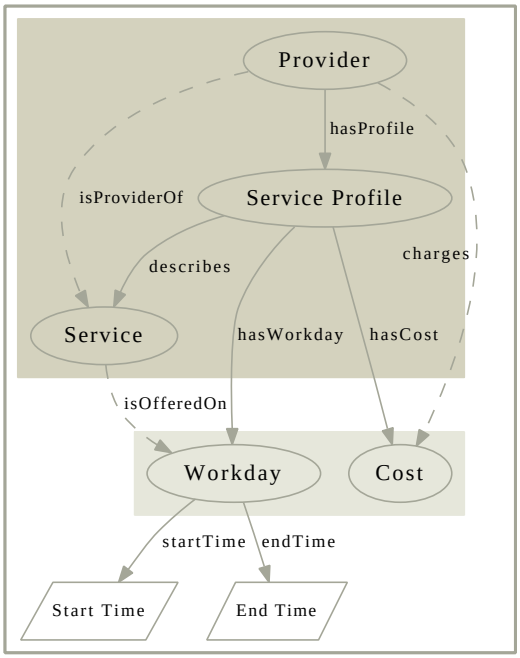

Figure 3. Service Provider Ontology

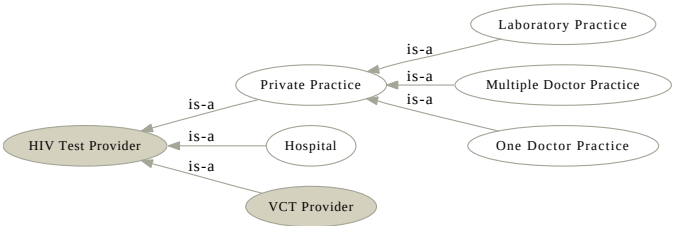

Figure 4. Inferred Model for HIV Test Providers

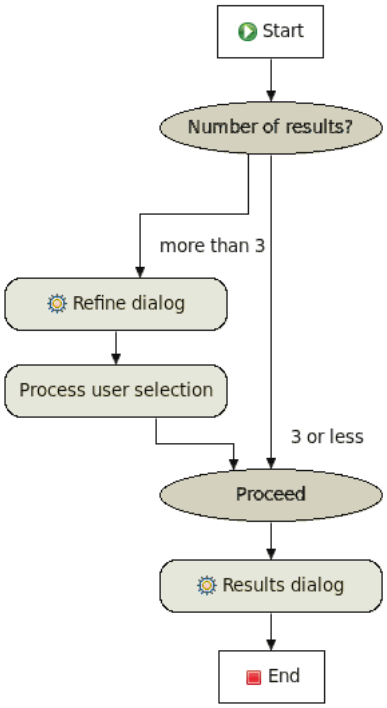

Figure 5. Process flow for generating VoiceXML dialogs

This process as aptly suggested by Noy and McGuinness [7] is non-trivial. It is iterative and demands careful analysis, feedback and redesign through each stage of the development life cycle.

In this paper, we will not discuss the construction process in detail. Instead, we will discuss the pertinent details of the ontology of discourse, shown in Figure 3. As the figure shows, the service provider ontology contains concepts such as *Provider*, *Service Profile*, *Service* and *Cost*. These concepts are related to each other via binary relations such as *hasProfile, describes* and *isProviderOf*. Also shown is that binary relations can exist between concepts and attribute values. In the figure, this idea is represented by binary relation *startTime,* which links concept *Workday* to a start time value, represented in the figure using a parallelogram.

The figure also illustrates that there are at least two types of binary relations: simple and complex relations. The former, represented by solid arcs, indicate that two concepts are directly related to each other. While the latter, represented by dashed arcs, indicate an indirect relation between connected concepts that can be established via a deductive process. For example, *isProviderOf* is a composition of *hasProfile* and *describes* (i.e. *hasProfile* o *describes*). This means that if some provider X has profile Y that describes some service Z, then it can be deduced that X is provider of Y.

Following the construction of the ontology, the next step was to instantiate it. That is, add instances (individuals) and their attributes to produce a working knowledge base (since without instances the concept of knowledge base does not exist).

The next vital step was to build a query interface. Essentially, this interface is a small program that acts as an intermediary between the ontology knowledge base and a reasoner. The purpose of the reasoner is to perform tasks such as checking for consistency and answering queries. We chose Pellet [8] reasoner for two reasons. First, at the time of construction, it showed a high degree of conformity with the (then) latest version of Web Ontology Language (OWL 2). Secondly, reports indicated good performance for many use cases [8].

Even though Pellet does perform reasonably well, we decided to increase the overall efficiency of query answering by disallowing queries that were not relevant to the core objectives of the voice application. Effectively, this translated in grounding the operation of the interface, but this was not deemed to be a negative thing; given that with voice, it is not possible to present information in parallel.

To put into context how this grounding effect was achieved, in loading the ontology, only the inferred model relating to HIV test providers is loaded into the working memory of the reasoner. This model can be visualised as a taxonomy tree, as shown in Figure 4. From the perspective of the interface, all the answers are captured within this tree. Therefore, the act of answering any query, amounts to traversing the tree using algorithms provided by the reasoner. It should be noted that based on some specified criteria, the reasoner can reconfigure the tree as part of refining an answer.

B. Implementing Dialog Processing with rules

When a query is made, it is typically not known how many results will be returned. For example, the query could generate 101 results. The question then was how best to

present these results to the user; given the serial nature of presentation imposed by voice communication.

The answer was to either present the results in batches to the user or find some criteria that could be used to shorten the list. We decided to use the latter option since it was more aligned to the objective of helping users locate a site that best fits their individual needs. Based on this decision, we generated a dialog flow (see Figure 5) to capture how the implementation would be achieved. As shown, if the number of results produced is less than or equal the thresh-hold value, which is three in this case, the results would be presented to the user i.e. a VoiceXML page for the results would be generated and delivered to the user. Otherwise a VoiceXML page would be generated to ask the user for criteria to be used to refine the results. After the user has provided an answer, the results would then be refined and presented to the user.

For all the pages generated, StringTemplate [9] was used. StringTemplate is a rendering engine that distinguishes itself by its ability to enforce separation of generation logic from the output format.

Processing the results with provided criteria could have been achieved by matching criteria with *if-then* statements. However, current trends are moving away from imperative to declarative programming, where rules are used. The former focuses on 'how to perform a task' while the latter focuses on 'what to do'. As cited in [1], the primary benefit of this shift from *if-then* to *when-then*, lies in the ability to separate knowledge (business logic) from implementation in manner that is readable and easy to update by non-technical users. This is important in that a change in the knowledge imposes no change in the source code, and as such, maintenance costs are reduced. Further, there is no need to involve the programmer in implementing changes.

Given the benefits, the next question was which rule engine to use? A number of alternatives were considered but after due consideration, we decided to use Drools (JBoss Rules) [1] based on evidence of good community support and the fact that is free.

In line with the dialog flow in Figure 5, the rule engine is invoked to process user selection after the user has submitted their criteria for refinement. When this happens, the rules stored in repository are fired and matched against the criteria. Following this, the refined results are presented to the user.

```
1  rule "Match criteria 1 - Alphabetic sort "
2    salience 3
3    when
4      $c:Criteria(userSelection==Criteria.CRITERIA_1)
5      $q:QueryResultMap($map:resultMap)
6    then
7      storeResultsMap = new TreeMap($map);
8      $q.setResultMap(storeResultsMap);
9      $q.setRefined(true);
10 end
```

The rules authored for use in Drools are represented in form shown above. As shown, the example rule matches criteria 1, which is alphabetic sort. This rule is matched when user selection is *CRITERIA_1* and the query result map (*QueryResultMap*) exists. When these conditions are satisfied, the rule fires and the contents of the query result map are put into a tree map, which performs the required sorting. The sorted results are stored in the *storeResultsMap* variable. Within the engine, the instance of the query result map (*$q*) is updated with stored results map and the refine variable is set to true.

VI. Testing and Analysis

As part of the development process, testing was performed to ensure that individual components of the system were coded correctly. Once the development was completed, we performed *system testing*. As highlighted in literature [10], [11], the key objective of system testing is to verify that all important requirements are met. These requirements can be classified as either functional or non functional.

Functional requirements describe 'what a system must to do' while non functional requirements describe 'how well the system must carry out its tasks' [12], [13]. Both types of requirements are important. However, in our case, system testing focused mainly on verifying and validating functional requirements. This is because our primary goal in building the system was to validate and demonstrate functional viability of the architecture that we proposed for developing voice applications.

Indeed we did consider performing other types of tests; because as suggested in literature [10], [11], focusing on a single type of testing is often not enough. For example, we considered performing extensive performance tests, but due to budgetary constraints, we could not. (As stated before, we used I6NET's VXI* VoiceXML browser. This browser is licensed on a per port basis and as such, there are cost implications for performing volume and stress tests.)

We also considered performing user acceptance testing but opted not to. We felt the complexity of the system was too low. For one, it is DTMF (dual tone multi frequency) driven and for another, it is a simple directory assistance service that provides one piece of information. Therefore, even though the system was 'powered' by a rich back-end, we surmised that, it would be difficult for users to differentiate the experience when the service was erected on the proposed architecture from when a traditional architecture was used. (Because, presumably, resource utilisation would be optimised for both architectures. Therefore, if, for example, a satisfaction survey was to be administered, the variance in the results would be insignificant.)

With regards to how we performed our system testing, we used a use case driven strategy. Essentially, we used the use case shown in Figure 1 to create a check list for verifying that key functionality of the system worked. This entailed using various testing techniques. For example, outcome prediction to determine accuracy of returned query results. In the event of a mismatch between the actual and predicted outcome, we used error guessing to hunt out bugs.

Given the stated low complexity of the system and the limited number of inputs, using these techniques proved to be effective. One valuable lesson learnt from the process was to appreciate the impact that the design of an ontology has on the overall functionality of system. Thus, for example, a problem with response times may be caused by insufficient system resources or a decision made in the design of the ontology. When dealing with knowledge representation formalisms like ontologies, this is important because as aptly noted by Davis *et al.* [14], knowledge representation and reasoning are inextricably entwined together. Thus, representation is tied to efficiency in computation and to what

constitutes intelligent reasoning.

VII. Conclusions

Although voice applications may have a limitation of presenting information serially, they offer a powerful alternative for contexts where visual modality may not be appropriate. For this reason, we believe that the development of these applications will continue to rise. The question, however, is how the development will evolve?

In this paper, we discussed implementation details of a system used to illustrate an architecture we have proposed for building voice applications. Based on this architecture, we believe that voice applications in the future will use inference engines, rules, ontologies and any other tool or resource that contributes to intelligent answering of user requests.

Acknowledgement

The authors would like to acknowledge the financial support of Telkom SA, Comverse SA, Stortech, Tellabs, Amatole Telecom Services, Bright Ideas 39, and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

References

[1] P. Browne, *JBoss Drools Business Rules*. Packt Publishing, 2009.

[2] G. Dragan, D. Dragan, and D. Vladan, *Model Driven Architecture and Ontology Development*. Springer, 2006.

[3] T. R. Gruber, "Towards Principles for the Design of Ontologies Used for Knowledge Sharing," in *Formal Ontology in Conceptual Analysis and Knowledge Representation* (N. Guarino and R. Poli, eds.), (Deventer, The Netherlands), Kluwer Academic Publishers, 1993.

[4] M. Maema, A. Terzoli, and L. Dalvit, "An Ontology-based Telephony Service for the Provision of HIV/AIDS Information," in *11th Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*, 2008.

[5] J. Ferrans, B. Porter, S. Tryphonas, B. Lucas, P. Danielsen, D. C. Burnett, A. Hunt, S. McGlashan, K. Rehor, and J. Carter, "Voice Extensible Markup Language (VoiceXML) Version 2.0," W3C Recommendation, W3C, Mar. 2004. Online: http://www.w3.org/TR/2004/REC-voicexml20-20040316/ [5 May 2008].

[6] M. Tsietsi, A. Terzoli, and G. Wells, "Mobicents as a Service Creation and Deployment Environment for the Open IMS Core," in *Southern African Telecommunications, Networks and Applications Conference (SATNAC)*, 2009.

[7] N. Noy and D. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology," tech. rep., Stanford University, 2001.

[8] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A Practical OWL-DL Reasoner," *Journal of Web Semantics*, vol. 5, no. 2, pp. 51–53, 2007.

[9] T. J. Parr, "A Functional Language For Generating Structured Text," tech. rep., University of San Franscisco, 2006. Online: http://www.cs.usfca.edu/~parrt/papers/ST.pdf [Last accessed: 5 March 2010].

[10] B. Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. New York, NY, USA: Verlag John Wiley & Sons, Inc., 1 ed., 1995.

[11] R. Black, *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. New York, NY, USA: John Wiley & Sons, Inc., 2007.

[12] P. L. H. Eide, "Quantification and Traceability of Requirements," tech. rep., NTNU Norwegian University of Science and Technology, 2005.

[13] D. Leffingwell and D. Widrig, *Managing Software Requirements: A Use Case Approach*. Pearson Education, 2003.

[14] R. Davis, H. E. Shrobe, and P. Szolovits, "What is a Knowledge Representation?," *AI Magazine*, vol. 14, no. 1, pp. 17–33, 1993.

Mathe Maema completed her BSc(Hons) at Rhodes University. She is currently reading towards an MSc degree at the same institution. Her research interests are in the area of telephony and knowledge management.

Alfredo Terzoli is Professor of Computer Science at Rhodes University, where he heads the Telkom Centre of Excellence in Distributed Multimedia. He is also Research Director of the Telkom Centre of Excellence in ICT for Development at the University of Fort Hare. His main areas of academic interest are converged telecommunication networks and ICT for development.

Lorenzo Dalvit is a lecturer in Education, where he is responsible for Information Communication Technology (ICT), both at an academic and practical level. He also works in close collaboration with the Telkom Centre of Excellence in Distributed Multimedia; with the South Africa-Norway Tertiary Education Development Programme and with the NGO Translate.org.za.