Tools Used While Developing Auracle: A Voice-Controlled Networked Instrument

Kriszán Varnó
Akademie Schloss Solitude
Solitude 3
D-70197 Stuttgart, Germany

Jason Freeman
Columbia University
2960 Broadway
621 Dodge Hall
New York, NY 10027

C. Ramakrishnan
Akademie Schloss Solitude
Solitude 3
D-70197 Stuttgart, Germany

ABSTRACT
Auracle is a networked sound instrument controlled by the voice. Users jam together over the Internet using only a microphone. Throughout the development process, the authors experimented with different approaches to interpreting vocal input and facilitating user interaction. This paper outlines some of the tools used to implement and evaluate those ideas, simulate the wide range of activities of Auracle users, and facilitate the development process.

Categories and Subject Descriptors
D.2.6 [Software Engineering]: Programming Environments – programmer workbench.

General Terms

Keywords
Auracle, Interactive Music Systems, Java, JSyn, Max/MSP, Network and Control, Open Sound Control, SuperCollider, Transjam, Voice Controlled Synthesis, Wire

1. INTRODUCTION
Auracle is a new networked sound instrument conceived by Max Neuhaus and realized by the authors. Users interact with each other in real time over the Internet, playing synthesized instruments in a “jam session.” Each instrument is entirely controlled by a user's voice, taking advantage of the sophisticated vocal control which people naturally develop learning to speak.

The program runs as a Java applet within a web browser. To control the instrument, users input sound into a microphone. Their sound is then analyzed, reduced into control data, and sent to a central server. The server broadcasts that data back to all participating users. Each client computer receives the data and uses it to control a software synthesizer (see Figure 1).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MM '04, October 10–16, 2004, New York, NY, USA.
Copyright 2004 ACM 1-58113-893-8/04/0010...$5.00.

Figure 1. Auracle System Architecture

2. ARCHITECTURE
The initial low-level analysis of the voice is performed through Linear Prediction (LP) analysis and tracking zero-crossings [11]. The microphone input is segmented into 40 ms frames, from which the fundamental frequency, the first two formant frequencies and bandwidths, the voicedness, and the root mean square (RMS) amplitude are computed.

The mid-level analysis parses the incoming data into gestures by looking for silences longer than a threshold. It analyzes each gesture to extract a feature vector of 43 statistical parameters. These statistics track the minimum, maximum, mean, and standard deviation of low-level analysis data across the length of a gesture, as well as information about the first derivatives and about the relationships between silent and non-silent frames within a gesture. The high-level analysis uses Adaptive Principal Component Extraction (APEX) to classify the feature vector for each gesture in terms of three high-level features [6].

Breakpoint envelopes created from the gesture’s low-level analysis data, along with the high-level feature values and the entire feature vector, are sent to a central server running TransJam, a Java server for distributed music applications [3]. By sending only control data, Auracle maintains low latency and high audio quality using a fraction of the bandwidth required for audio streaming. The TransJam server does not
process the data it receives but simply forwards it to all clients.

Each client receives the data from the server and passes it to a mapper, which turns the incoming data into synthesizer control parameters. For example, it could use the amplitude breakpoint envelope to control the amplitude and the “noisiness” of the synthesizer, or it could use the formant breakpoint envelopes to control a bank of bandpass filters. An audio effects unit then processes the mix of all active synthesizers. We use the JSyn API to perform the audio synthesis and effects processing [2].

The analysis data is transmitted to the server only once a complete gesture has been detected. This reduces network traffic and generally uses the network more efficiently. Data is only mapped onto synthesis control parameters when it arrives from the server, even when the data was created by the local client. This creates a short delay between the voice input and synthesized response; we have found that this latency is not a disadvantage but rather facilitates a conversational style of interaction which works quite well.

There are virtually an infinite number of ways to map and synthesize the data generated by the analytical components. We built many experimental mappers with different control

---

Figure 2. The Auracle GUI (used in the public release)

Figure 3. The Auracle Testbed GUI (used by developers only)
methodologies and created synthesizers exploring FM synthesis, physical modeling, additive synthesis, subtractive synthesis, and granular synthesis. Our final implementation incorporated the best aspects of each experiment, as determined by our own assessments and feedback from outside testers.

3. TESTBED DEVELOPMENT

3.1 Dynamic System Configuration

During the development of Auracle, we rapidly created many different analyzer, mapper, and synthesizer components. Auracle’s architecture uses Java interfaces, reflection, and the observer pattern, combined with an avoidance of direct cross references, so that components can be mixed and matched to form a complete system. During startup, the application reads a text file specifying the particular components to be used and instantiates the corresponding configuration.

Reconfiguration of Auracle does not require the source code to be recompiled, but it does require the configuration file to be edited and the program to be restarted. Rapid comparisons between configurations are not possible. And small tweaks to synthesizer parameters require changes to the source code; they cannot be specified in the configuration file. As the number of experimental components grew, tracking and comparing components and configurations became increasingly difficult.

To address these limitations, we created the Auracle Testbed, a separate application (see Figure 3) used only in the development process and not included in the public release (see Figure 2). Popup menus in the Testbed’s GUI select analyzer, mapper, synthesizer, and effects unit components, and sliders adjust internal synthesizer parameters for fine-tuning control.

The Testbed saves configurations of analyzers, mappers, synthesizers, and effects units as patches. Developers annotate patches through name and description fields to add comments or help explain them to other team members. The patches are saved as text files and also displayed as buttons in the GUI. A single button press switches to a different system configuration, enabling rapid comparisons between patches. The change in Auracle’s configuration is immediate; no text files need to be edited and the application does not need to be restarted.

From within the Testbed, developers can also easily upload patches to the group development server to share them with other team members, who can use them in a group “jam session” or download them to their local machine.

3.2 Integrating Existing Tools

The Auracle Testbed can also send analysis data to any application which supports the Open Sound Control (OSC) protocol [13]. We used this feature to send Auracle data in real time to SuperCollider [9], Max/MSP [5], and Wire [4]. By combining Auracle with external sound development tools, we were able to quickly prototype new ideas using existing synthesis libraries and user-friendly environments. These external programs also enabled us to develop during runtime; we could hear synthesis algorithms responding to Auracle data even as we wrote them.

We exported synthesis patches developed in Wire as Java source code and directly integrated them into Auracle’s Java source tree. For synthesis algorithms designed in the other applications, we manually ported the most successful algorithms to Java, which was straightforward.

4. AUTOMATING USER INPUT

4.1 Simulating a Single User

Since Auracle is a voice-controlled instrument, we needed to constantly create vocal sounds in order to evaluate and test it. But it was difficult for a single developer to simulate the full range of vocal sounds expected from actual Auracle users, and constant vocal production grew fatiguing.

During the development of Auracle, we recorded 230 vocal gestures as sound files. The collection includes gestures from ten participants, half male and half female, who come from seven different countries and speak six different native languages. The sounds range from short clicks to spoken words to breathing sounds to hummed melodies. The collection helped us to work more efficiently and to simulate a wider range of user inputs than we could have otherwise.

A second, smaller collection of sound files documented gestures which caused problems such as inaccurate analyses, overloaded synthesis filters, or even crashes. We used these files to consistently reproduce problems as we were trying to fix them.

The Testbed enables us to use these collections as input into Auracle; a popup window provides controls to select, trigger, and loop sound files.

4.2 Simulating Multiple Users

Auracle is designed for use by an ensemble of participants, so it was important to test it in group situations throughout the development process. Mapping and synthesis components sounded dramatically different when used individually than when used in a group “jam session.” Many bugs only occurred in group situations. And we also needed to test the server under heavy loads to benchmark performance and determine capacity.

To address these needs, we developed a Headless Client to simulate the activity of a single user. In order to reduce CPU usage, the Headless Client pre-analyzes audio files and stores data in a form ready to transmit to the server. It references this preprocessed data when “jamming” on Auracle.

A command-line application launches several Headless Clients simultaneously to simulate one or more ensembles of participants. A developer can simulate dozens of users from a single machine and then launch a single instance of the complete applet to “jam” with them interactively.

5. DEBUGGING MECHANISMS

5.1 Automated Testing

We used EXtreme Programming (XP) methodologies [1] in the Auracle development process to maintain high code quality. JUnit [7] unit tests, many of which incorporated the Headless Client, ran nightly on the server.

Auracle development was rapid, and data structures and code organization changed dramatically from week to week. Automated tests ensured that all existing components continued to function even as the architecture changed. As the application moved closer to a public release, we wrote
additional unit tests to track performance and test code optimizations.

5.2 Tracking User Sessions
Once we released a beta version of Auracle to the public, we wanted to monitor user activity to identify the problems users encountered. A combination of several different logging mechanisms track this information.

We use Analog [12] to analyze Web server logs and obtain basic information about site visitors. And the TransJam server tracks the number of users of the Auracle applet, the length of each user session, and whether users encountered other participants online or used Auracle alone.

To complement this information, the Auracle applet maintains more detailed logs on the server. The applet uploads data via an HTTP post; a PHP script then processes it and stores it in a MySQL database. Along with basic user information, the database logs the client's operating system, web browser, and Java implementation. It also includes any client-side error messages generated during the user session, along with Java stack traces of exceptions.

This logging data helps us more easily track and fix bugs. When users send us problem reports, we can quickly locate their session in the database and find information about their system configuration and any errors which Auracle logged; they do not need to figure out these details themselves. We can also look directly in the database to find errors which were never reported by users at all. Often, a stack trace in the log points us to a specific line of source code and an easy solution.

The database is accessible to team members via phpMyAdmin [10], a web-based interface to MySQL. We can browse or search the database, execute more advanced SQL queries, or export data in standard formats for analysis in other software. These powerful search capabilities help us track patterns in errors, reduce debugging time, and ultimately make Auracle more stable.

6. CONCLUSION
Auracle team members live in California, Arizona, Germany, and Italy, so most of our collaboration was done remotely. Server-side tools, such as Testbed patch sharing, automated nightly testing, and a central error database, enabled us to work together more effectively.

Custom development tools, such as the Auracle Testbed and Headless Client, are usually used by a small audience and often lack the polish of a finished product. It is time consuming to create documentation and support for these custom applications. When the team using the application is in remote locations, simple questions can be tedious to resolve by e-mail and telephone.

By integrating Auracle with existing commercial and open-source tools, we were able to minimize the functionality required in these custom development applications. With OSC support, each team member could develop synthesis algorithms for Auracle using familiar applications with polished graphical interfaces and sophisticated synthesis tools. With the JUnit testing framework, nightly testing was easy to automate. And with the use of a standard database program to store logging data, powerful search features were trivial to add. This approach enabled us to minimize the time spent building tools and maximize the time spent building Auracle itself.

7. ACKNOWLEDGEMENTS
The Auracle project is a production of Akademie Schloss Solitude with financial support from the Landessstiftung Baden-Württemburg. We express our gratitude at their generous support.

Auracle is available at http://auracle.org.

8. ADDITIONAL AUTHORS
Additional authors: Phil Burk (SoftSynth, contact: http://www.softsynth.com), David Birchfield (Arizona State University), Max Neuhaus.

9. REFERENCES