AUTOMATED TESTING OF OPEN-SOURCE MUSIC SOFTWARE WITH OPEN SOUND WORLD AND OPENSOUND CONTROL

Amar Chaudhary
amar@ptank.com

ABSTRACT

Providing robust systems for live musical performance has been a difficult problem. Such systems are highly dynamic with lots of paths for execution; even small changes in input can lead to different results. Comprehensive testing is difficult, tedious and resource intensive, more so for open-source projects that often lack the resources to do such testing. We present a system for efficient automated testing of the Open Sound World (OSW) open-source music environment using OpenSound Control (OSC) messages sent from Python scripts. The automated testing system helped the developers to release a significantly more robust version of OSW in 2004, and is an important tool in the development of OSW 2.0. OSW supports VST and LADSPA plug-ins as is binary compatible with most Pd externals. Thus, the automated testing system in OSW can be used to test these external plug-ins as well.

1. INTRODUCTION

Live musical performance with computers requires robust software over a large set of changing user input. For more complex systems, the likelihood that a small change in user input between performance and rehearsal (e.g., a missed note, a timing glitch or an input value that is out of the tested range) could have a serious effect on the system increases. Even well-designed systems that employ techniques from computer science to avoid faults are subject to issues arising from bugs in the code or the inherent unpredictability of complex dynamical systems. Such systems require comprehensive testing to minimize the risk of faults.

Such testing is a costly and complex process. Commercial software developers often apply large amounts of time and resources to testing; open-source projects without such resources must often rely on the goodwill and expertise of their user base to report individual issues. Even when rigorous testing is undertaken, it is more often than not a tedious process that provides little technical or intellectual reward for the open-source developer or musicians using his or her software.

Automated testing has been used in many projects to provide efficient testing. Open-source software, such as the GNU compiler collection (GCC) [4], often include test scripts invoked with the command "make test." However, such batch testing becomes more challenging when applied to a reactive real-time system such as software used in musical performance.

The OpenSound Control (OSC) protocol [9] provides a framework for automated testing in real-time music applications. A large number of test cases, written as sequences of OSC messages, can be executed quickly. Bi-directional OSC communication provides a mechanism for feedback on success or failure to the testing agent, and OSC queries can be used to dynamically adapt testing to the state of a system (e.g., extensions that may or may not be installed).

This paper describes an effort to use OSC to build an automated testing system for Open Sound World, an open-source environment for real-time music and multimedia applications.

2. BACKGROUND

Open Sound World (OSW) is large, feature-rich system for developing real-time music and multimedia applications to use in live performance or other interactive settings [2]. It has been available to the public as open-source software since 2001. One of the goals during the development of OSW 1.2 in 2004 was significantly increasing the robustness of the application and trying to eliminate bugs before they became part of public releases. In addition to the challenges for open-source testing described in the previous section, OSW has the added challenge of running reliably on diverse platforms: Windows 2000/XP, Linux and Mac OS X. We are currently developing a new version based on feedback and contributions from developers and users. OSW version 2.0, to be released in 2005, builds on the original design and goals of the project by improving important aspects of the system, such as user interface, distributed client-server architecture and greater compatibility with other languages and environments. As with the development of version 1.2, this release presents a significant testing effort in the presence of limited testing resources.

These challenges make the OSW project a strong candidate for automated testing. Additionally, OSW supports bi-directional OSC communication and OSC queries. This suggested that we develop a strategy based on OSC messages for multi-platform automated testing.

3. OSC SUPPORT IN OSW

This section briefly describes several features of the OSC-server implementation in OSW that were necessary for the development of our automated testing system.

3.1. All objects in OSW are OSC addressable

In OSW, components called *transforms* are combined to form programs called *patches*. Patches are themselves transforms that can be included in other patches. This gives rise to a hierarchical name space in which every transform has a unique address, which be used as an address for OSC messages. Additionally, the inlets, outlets and state variables (i.e., internal variables and parameters of a transform) are themselves named components with unique addresses. Thus an OSW program forms a complete OSC address space where any object (patch, transform, variable) can be queried inspected or modified using OSC messages.

3.2. Support for OSC Queries

OSW includes a full implementation of OSC queries, which was largely the work of Andrew Schmeder at Center for New Music and Audio Technologies (CNMAT) [8]. OSW supports queries of objects for documentation, type signatures, current values and contents (e.g., state variables of a transform, or transforms in a patch). Clients can use queries to discover the dynamic address space of OSW programs. A client can then use OSC to send any message understood by any of the objects in the programs.

3.3. Every message receives a reply

Every message received by an OSW server results in a return message to the client. This allows a client to detect a communication or server failure by timing out on reply. Additionally, the reply message always has the same address as the original message:

/patch/output0/sample_rate/current-value
→ /patch/output0/sample_rate/current-value 0 44100.0

In the above example, a query was sent to get the current value of the state variable /patch/output0/sample_rate. The reply included the query address, an error code of zero (indicating success), and the value itself. If the variable was not found on the server, an error would have been returned

3.4. Programming via OSC messages

OSW programs can be built or modified remotely using OSC messages. Patches and transforms can be instantiated, and connections established between inlets and outlets. For example, the following message:

/patch/add-transform "osw::Sinewave" "sin0"

will instantiate a transform named sin0 of class Sinewave in the patch named /patch. To connect the outlet of this transform to an existing audio output named output0, the following message is used:

/patch/sin0/samplesOut/connect "/patch/output0/mix"

4. DEVELOPING THE TEST SYSTEM

A Python-based testing framework was developed to send and receive OSC messages as described in the previous section, and evaluate the return messages against expected results. The framework relied on the `unittest` library in Python [7]. In this framework, a series of *test cases* are aggregated into a *test suite* (these terms are standard in software-quality-assurance organizations), which can then be automatically executed. Each test case consists of one or more individual tests, which are scripts written by a test developer.

Individual tests were organized into test cases by conceptual areas, for example, one test case included several tests for list processing, while another dynamically build simple patches to output audio. We present one example test from the list-processing test case `TestListServer`. In the example test, the OSW transform list::Reverse is instantiated and used to process two different lists: one numeric and one that contains strings and a nested list. For each message sent to the server, the reply is first tested for validity using the helper method `errorTest`. If a reply is not received or contains an error condition, the test fails. Once the list-reversing transform is instantiated, it can be tested by setting the value of its inlet to a particular list to be tested. Setting the inlet in OSW will *activate* the transform and process the input. The outlet of the transform will then contain the result. The test script then sends a "current-value" query to the outlet to obtain the result, which is compared to a "correct result" generated by the script (we assume that Python's list-reversal operation has itself been tested enough to be considered correct). If the results are equal (within a certain round-off tolerance for floating-point numbers), the test is considered successful.

```
class TestListServer(unittest.TestCase):

        …
   def test_Reverse(self):
        """Reverse lists"""
        #load the external
         TestSupport.runOSCCommand('/load-
        osx',['list/Reverse'])

        #instantiate the transform
         result =  TestSupport.runOSCCommand
               ('/patch/add-transform',
               ['list::Reverse','reverse1'])
         self.errorTest(result)

        #first list to try
         l = [1,2,2.718281728459,
                3,3.14159265358979323846,4,5]
        #send list to the transform inlet
        #this will trigger the transform to run
         result = TestSupport.runOSCCommand
              ('/patch/reverse1/listIn',l)
         self.errorTest(result,len(l))
        #get the result from the output
         result = TestSupport.runOSCCommand
   ('/patch/reverse1/listOut/current-value',[])
         self.errorTest(result)
        #is the list the correct length?
         self.assertEqual(len(result),len(l)+3)
        #OK, now compare to the same reversal
        #  in Python
         r1 = result[3:]
         r1.reverse();
         for (a,b) in zip(r1,l):
             self.assertAlmostEqual(a,b,6)

        #let's try a more complex list
         l = ['apple','pear',[1,2,3],'banana']
         result = TestSupport.runOSCCommand
                ('/patch/reverse1/listIn',l)
         self.errorTest(result,len(l))
         result = TestSupport.runOSCCommand
('/patch/reverse1/listOut/current-value',[])
         self.errorTest(result)
         self.assertEqual(len(result),len(l)+3)
        …
```

Note that "success" of a test script means that it ran to completion without triggering any of the assertions along the way.

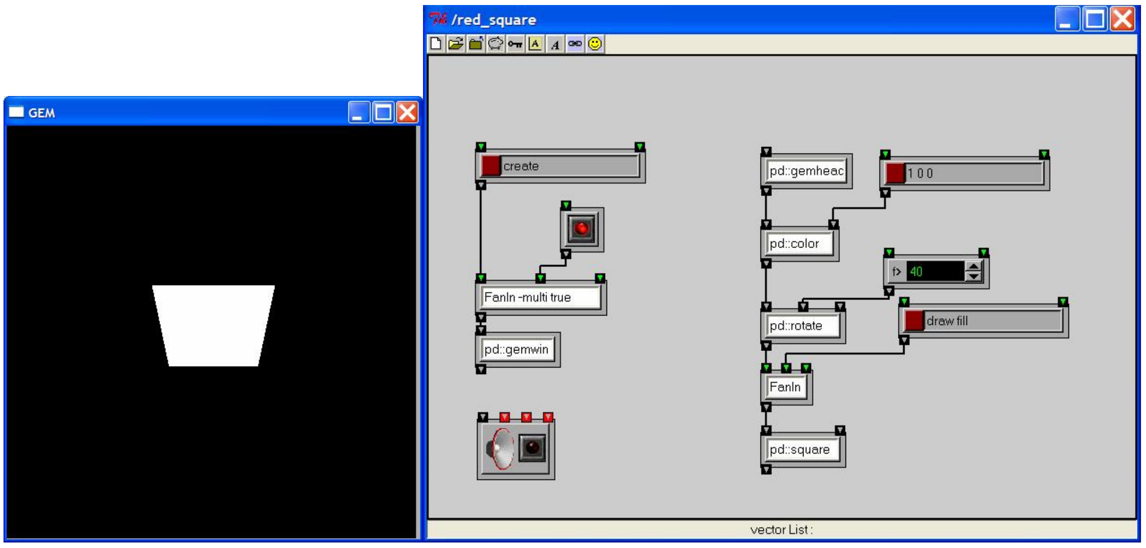

Figure 1. GEM Pd externals running in OSW 1.2.x. The Pd externals (prefixed with "pd::") accept input from native OSW transforms, including lists from MessageBox transforms that are interpreted as Pd messages.

4.1. Testing of DSP functions

Similar to the list-processing example above, DSP transforms that operate on audio input can be tested by sending a known block of samples to audio inlets and comparing the results with known correct values. OSW includes an extension to OpenSound Control that recognizes signals as blocks of integer or floating-point values.

Synthesis transforms that generate audio output in response to virtual time input (i.e., generating samples every clock tick), OSW includes an OSC method /advance-clock that advances the main clock one unit (which is a function of the current audio sample rate and block size). Advancing the main clock, which is a standard OSW state variable, will activate all transforms that accept time sources, including synthesis transforms such as Sinewave. The output values of these transforms can then be obtained to test for correctness. This process allows the testing system to evaluate the correctness of signal-processing transforms in OSW without being subject to the performance constraints and potential non-deterministic behaviour of real-time processing, i.e., the clock is only advanced one tick and the output values are available until the testing system explicitly advances the clock again.

5. TESTING COMPATIBLE PLUG-INS

OSW includes support for several "standard" plug-in architectures, including VST, LADSPA and Pd externals. Although such plug-ins are developed outside the context of OSW, developers of these plug-ins can use them in the context of OSW to develop OSC-based automated test suites. This section describes standard plug-in facilities in OSW and how the automated testing system in OSW can enhance reliability in external software.

5.1. VST and LADSPA plug-ins

VST plug-ins have become a de-facto standard for synthesizers, effects and other musical components. A wide variety of VST plug-ins are available both freely and commercially. Additionally, several popular commercial and open-source music packages can host VST plug-ins and integrate them into the applications functionality. Similarly, LADSPA plug-ins are becoming a de-facto standard for audio plug-ins on Linux, as well as for open-source audio development on other platforms (e.g., Audacity supports LADSPA plug-ins on multiple platforms [1]). The remainder of this section, however, focuses on VST.

OSW can host VST plug-ins via the Vst transform. This transform loads a VST plug-in and exposes its audio-signal and MIDI I/O as inlets and outlets. Additionally, VST "automation parameters" (parameters of the plug-in that can be read or modified by the host) are exposed as transform inlets or state variables. As variables of an OSW transform, these parameters become OSC addressable and can be queried or modified by external OSC hosts. Additionally, OSW supports atomicity on VST-parameter updates via OSC bundles. Because all parameters of the VST plug-in are OSC addressable, automated test suites can be developed similar to those described in section 4. Thus, even in cases where it not considered an end-user target by the plug-in developer, OSW and OSC can become valuable testing and debugging tools for developing more robust plug-ins.

5.2. The "Pd-compatibility Layer"

A large number of externals for Max/MSP and Pd [6] have been developed over time and used extensively in the computer-music community. Once a piece has made use of such externals, it usually requires that it remain realized in that environment (which becomes a greater challenge as programming environments mutate or become unsupported). Hand-porting of complex pieces to newer systems may prove difficult or even impossible without access to features of the original system.
We have taken steps to address this issue by providing a "Pd-compatibility layer" for OSW. This system provides a set of

intrinsic Pd objects as OSW transforms as well as the ability to load and use most Pd externals as transforms without recompilation. For example, the library for the popular Pd external fiddle~ can be loaded into OSW via the compatibility layer, and OSW users can then instantiate the transform `pd::fiddle~` to use it in their patches. 1 A large number of Pd objects have been successfully imported into OSW for use in projects, including PeRColate [5] and the GEM library [3], as illustrated in figure 1.

Similar to hosted VST plug-ins, the named inlets and outlets of Pd objects instantiated within OSW become OSC addressable, allowing remote control and access of Pd objects without specialized patches and infrastructure for receiving and interpreting OSC messages. In addition to unit testing of individual Pd objects via OSC messages, entire music programs or pieces for live performance can be ported to OSW using many of the original Pd objects. Thus, entire pieces can be tested extensively using automated scripts prior to rehearsal and performance.

6. DISCUSSION

Initial use of the automated test system in the summer of 2004 uncovered numerous issues. Systematic instantiation of every transform identified basic issues in a few transforms that we used less frequently in development or musical practice, such as missing parameters leading to instantiation errors, or various transforms affecting each other via shared resources that may be have been locked or contaminated. The extensive set of mathematical transforms in OSW was systematically tested over a range of input, uncovering several mathematical errors. All of these bugs were relatively easy to fix, but would have been difficult to detect and isolate with ad-hoc user testing in the field.

Additional blocks of test cases have been developed for basic audio performance, DSP and OSC-based control of documentation and tutorial patches bundled with OSW. This latter set of tests in particular was useful for isolating issues in complex interactions between transforms, such as unprotected resources, and systematically testing behavior of the system on different operating systems.

The end result was a significantly more robust release for OSW 1.2. Although the data for a rigorous comparison is not currently available, anecdotal evidence suggests fewer "functional bugs" or application faults during normal use. The most common issues now reported by users are system incompatibilities, which our current set of tests have not addressed.

The testing framework and test cases are now part of the source release of OSW at http://osw.sourceforge.net and available for users to perform and modify tests on their own systems.

[1]Although OSW does not use the convention of tilde characters for "signal" objects found in Max/MSP and Pd, the suffix is preserved on imported Pd objects.

7. ACKNOWLEDGMENTS

The author wishes to acknowledge to contributions of Andrew Schmeder in developing the OSC infrastructure in Open Sound World, as well as the support of the Center for New Music and Audio Technologies at UC Berkeley, and the UC Discovery Grant.

8. REFERENCES

[1] Audacity. http://audacity.sourceforge.net

[2] Chaudhary, A., A. Freed and M. Wright. "An Open Architecture for Real-time Music Software." *Proceedings of the International Computer Music Conference*, Berlin, 2000.

[3] Danks, M. "Real-time image and video processing in gem." *Proceedings of the International Computer Music Conference,* Thessaloniki, Greece, 1997.

[4] The GNU Compiler Collection. http://gcc.gnu.org/

[5] PeRColate library. http://www.akustische-kunst.org/puredata/percolate

[6] Puckette, M. "Pure Data: Another Integrated Computer Music Environment." *Second Intercollege Computer Music Concerts*, Tachikawa, Japan, 1996.

[7] Purcell, S. Python Unit Testing Framework. http://pyunit.sourceforge.net/pyunit.html

[8] Schmeder, A. W. "Implementation of a Scalable and Dynamic Interface for Open Sound World using OpenSound Control." http://www.a2hd.com/research/osw-osc/osw-osc-03.php

[9] Wright, M. OpenSound Control Specification version 1.0. March 26, 2002. http://cnmat.berkeley.edu/OpenSoundControl