An Automated Testing Suite for Computer Music Environments

Nils Peters
ICSI, CNMAT UC Berkeley
nils@icsi.berkeley.edu

Trond Lossius
BEK
trond.lossius@bek.no

Timothy Place
Electrotap
tim@electrotap.com

ABSTRACT

Software development benefits from systematic testing with respect to implementation, optimization, and maintenance. Automated testing makes it easy to execute a large number of tests efficiently on a regular basis, leading to faster development and more reliable software.

Systematic testing is not widely adopted within the computer music community, where software patches tend to be continuously modified and optimized during a project. Consequently, bugs are often discovered during rehearsal or performance, resulting in literal "show stoppers". This paper presents a testing environment for computer music systems, first developed for the Jamoma framework and Max. The testing environment works with Max 5 and 6, is independ from any 3rd-party objects, and can be used with non-Jamoma patches as well.

1. INTRODUCTION

1.1 Testing in sound and music computing

Stability and reliability is a general and important concern in all development and use of software applications. To artists and musicians working with real-time media processing environments such as Max, SuperCollider or Csound, programming is an integral part of their artistic practice. Their patches can be considered software programs, and they also become critical and integrated parts of the resulting artistic works, be that in the form of virtual audio-visual instruments for live performances, or patches used to run installations. In these contexts software reliability is not just a question of being able to work efficiently up front while preparing the artistic work, avoiding the frustrating experience of loosing time and work in progress due to sudden and unexpected bugs and crashes. The very presentation of the works in concerts, performances and exhibitions depends on the software, and quite literally software defects can be show stoppers.

In 2002 the National Institute of Standards and Technology (NIST) reported that software defects cost $59.5 Billion annually in the US, while a third of it could be eliminated by an improved testing infrastructure [1]. One believes that the earlier a bug is found, the cheaper the fix becomes. A systematic approach to testing is part of contemporary programming practice, making extensive use of solutions for running automated tests on a regular basis.

In the sound and music computing community adoption of systematic approaches to testing remain less widespread. Bugs often surface when making changes to the program, or the target environment or operating system, and for this reason many artists tend to be hesitant about changing their performance computer system or software version because of the fear of unforeseen problems. They might also be reluctant to doing last-minute changes and improvements to their patches. The programs or patches developed are often custom developed for a particular project, and with increasing complexity patches become increasingly vulnerable.

1.2 Importance of testing to Jamoma development

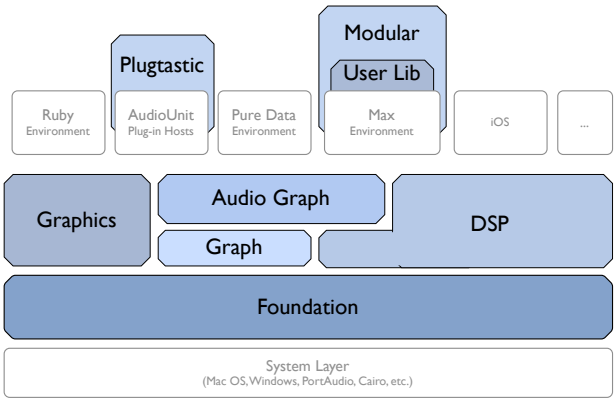

Figure 1. The Jamoma layered architecture

Jamoma is a real-time interactive media processing platform structured as a layered architecture of several frameworks (see Figure 1), providing a comprehensive infrastructure for creating computer music systems [A]. Jamoma Foundation provides low-level C++ support, base classes, and communication systems. Jamoma DSP specializes the Foundation classes to provide a framework for creating a library of unit generators [2]. Jamoma Graph networks Jamoma Foundation based objects into graph structures. Jamoma Audio Graph [3] is a C++ framework that extends and specializes the Jamoma Graph layer. It provides the ability to create and network Jamoma DSP objects into dynamic graph structures for synchronous audio processing. Jamoma Graphics provides screen graphics. Jamoma Modular provides a structured approach to development and control of modules in the graphical media environment Max [4]. Plugtastic, the latest addition to the Jamoma ar-

chitecture, aims to create VST and AudioUnit plugins from Max patches created with the Jamoma Audio Graph.

The Jamoma distribution is available for Windows and Mac OS with a BSD open source license. At the time being Jamoma is mainly targeted at Max, but prototype implementations are available for using parts of Jamoma with Pure Data, as AudioUnit plugins and on the iOS platform. Recently we have also started to explore the use of Jamoma on Beagle and Panda boards.

Jamoma has been in development for more than eight years, and has a mature, well-established codebase and a large, active development team. As can be seen in Figure 1 the higher-level frameworks such as Modular depends on several lower frameworks, and each of the frameworks contains shared code providing core functionalities for several Max externals. This potentially makes Jamoma vulnerable to the introduction of bugs and errors. For instance, a change to the code in Foundation can have far-reaching consequences, and might introduce issues and problems in all of the frameworks. Similarly there is a complex set of relationships between the Max externals that provide the core infrastucture of Jamoma modules, and a change to any of these externals might make all modules unstable. The set of functionalities and dependencies are far too extensive and complex to be able to test manually whenever code is being altered. Instead the Jamoma team strived to develop a structured solution for automated testing. This is used to implement a growing number of tests that aim at ensuring that new functionalities work according to specifications and that development do not introduce bugs.

2. AUTOMATED TESTING

The simplest approach to testing is to perform manual tests invented on the fly while developing. This has the disadvantage that test cases represent a valuable investment that will disappear after the testing has been completed. If tests instead are implemented as executable code, they can be saved and run again after changes to other components of the program. This is known as regression testing [5], and helps ensuring that resolved issues do not reappear later on.

Tests might be added at different stages of the development process. In test-driven development, the implementation of a new feature always starts out with development of tests, and the tests also serve to specify and document the feature. Tests might be developed explicitly to scrutinize the program for problems, and according to [5] "testing is the process of executing a program with the intent of finding errors." When bugs emerge in real-world use, the first step towards resolving the bug is to determine a sequence of steps by which to reproduce the bug and assess its output. Thus, any bug report should ideally be reported in the form of a test.

A number of guidelines for test development are offered in [6]: A test should have one and one only assert statement; we want to test a single concept only in each test function. Tests should be fast to execute, so that they can be run often. Tests should not depend on each other. Tests should be repeatable in any environment, as this will vastly help stability in cross-platform development. Finally the tests should be self-validating (have a boolean output).

Well-design and readable tests can be executed easily and often. This will speed up development and improve stability, and keep the code flexible, maintainable, and reusable. If you have tests, you do not fear making changes to the code, but without tests every change is a possible bug [6].

2.1 Forms of testing

A complete taxonomy of types of software testing is beyond the scope of this paper, and likely to be impractical for the scale of most computer music projects. Three forms of testing that cover the most essential cases are Function Testing, Unit Testing, and Integration Testing. These same three forms of testing are emphasized in other software disciplines such as web application development using Ruby on Rails [B] or Django [C].

The purpose for these different types of testing is to exercise different layers of complexity in the software using a structured and manageable approach.

2.1.1 Function Testing

Function testing exercises the various actions available through the application programming interface (API) of any given object (unit) of code. This test will query for available parameters, attempt to set them to random values, check that ranges are limited correctly, attempt to process audio, etc. One example of Function testing is the AudioUnit validation performed by Apple's `auval` command-line program [D].

2.1.2 Unit Testing

Where function testing will ensure behavioral conformance in a generic manner, Unit tests are custom tailored to examine the specifics of a given object or unit. Unit tests verify functionality of a module or specific section of the code. These tests are most often written together with the object. In Jamoma they are typically implemented as a method in C++.

One benefit of unit testing is bug localization inside small coding units. Unit testing alone cannot verify the complete functionality of a piece of software, but it is rather used together with Function testing to assure that the building blocks the software uses work independently of each other. Function and Integration testing, although very useful, have generally a much coarser localization (e.g. the single Max external).

2.1.3 Integration Testing

Integration testing works to expose defects in the interaction between any number of units when combined with each other. In Jamoma we implement this kind of test on the patcher level in Max, where many objects can be connected together into a larger operational system.

2.2 Testing in computer music

In computer music, audio processes are crucial, but designing DSP tests pose particular challenges as the data information extends over time, and the sample values of audio

vectors might not be as easily accessible as in the case for control data.

2.2.1 DSP testing

Traditionally, many audio devices are assumed to be linear time-invariant and are therefore described through their impulse response. Standard test signals are clicks, sine sweeps, or noise signals [7]. Three testing strategies based on comparing inputs to outputs of a device under test are discussed below.

In *bit-exact testing*, the output of a DSP process is compared bitwise with a pre-computed reference signal. For instance, the accuracy of a lowpass filter is tested by comparing the impulse response with the reference impulse response for that filter. The reference impulse response may have been captured from a reference implementation or has been computed based on the lowpass filter specification. Only if every bit of the output stream is exactly identical with the reference signal, is the test correct. In practice, bit-exact testing can be challenging, because of system- and platform-dependent truncation errors in floating point number calculations. Furthermore, the bit-exact test fails, even if the signal contains the correct information, when the output signal contains noise or a time-delay. In certain situations it can be also inconvenient or impossible to have a reference signal. For instance, for time-variant systems where the output is not exactly predictable e.g., stochastical artificial reverbs, bit-exact testing does not work. According to [8], bit-exact testing has been used for low-bit rate speech codecs.

A variety of *bit-exact testing* can be seen when testing floating point numbers within a given absolute or relative tolerance range. Since floating-point truncation errors differ across computer-platforms and compilers, *tolerance testing* is an efficient workaround especially for high-resolution audio signals, where the perceptual differences of the least significant bit may be not noticeable.

In *parametric testing*, rather than comparing signals with each other, signal features of the output signal are compared to signal features from a reference signal. A test passes if all tested features from both signals are equal within a pre-defined tolerance range. One example would be an examination of the RT_{60} reverb time in order to validate properties of an artificial reverb. Besides technical parameter, parametric testing allows accounting for perceptual properties e.g., for verifying the audio quality of a compression algorithm. One has to keep in mind that parametric testing depend on the truthfulness of the extracted features and that the quantity and kind of meaningful and necessary tests depends on the DSP process.

2.2.2 Prior work

In the mass production of audio devices such as microphones, loudspeakers and amplifiers, commercial testing environments e.g., those by Audio Precision [F] are widely used. An early report of automated testing in the audio device manufacturing process can be found in [9].

For Nokia's DSP Entertainment Audio Platform, [8] describes the development of an automated testing environment based on parametric testing.

In the computer music community however, automated testing seems to be not widely explored yet.

The Open Sound World (OSW) computer music environment used automatic testing based on sequences of OSC messages to communicate between a python-based testing framework and the OSW application [10]. In OSW, all objects are equipped with bidirectional OSC communication, which allows to configure, control, and query the system directly via OSC. Also using OSC, vectors of audio samples are communicated, thus enabling evaluation of audio processes within the python framework.

The Faust project [11] does not provide any testing feature, but allows for generating and storing of output signals and the generation of Matlab code for further unit testing [12].

3. THE JAMOMA TESTING SUITE

This section explains how unit and integration testing is done in Jamoma, as illustrated by the Jamoma dataspace library [13].

Most of Jamoma's Max externals are implemented as generic C++ units which are made available to Max by using a generalized wrapper function (see also [2]). The C++ functionalities are validated using unit tests, while testing of the Max externals are performed as integration tests.

3.1 Unit testing

Several testing frameworks exists for C++, such as CppUnit, UnitTest++, and Google Testing Framework. However, as Jamoma Frameworks are based on a dynamically-bound message passing model of communication and discoverability rather than statically linked method calls, these existing test infrastructures are inadequate to meet our demands.

In Jamoma Foundation we have created a general infrastructure to support running automated tests with various data types. For each class a test method is implemented that can be extended to add the relevant tests for the class. For instance `TimeDataspace.h` contains the following method:

```
virtual TTErr test(TTValue& returnedTestInfo);
```

This is defined in `TimeDataspace.cpp` as:

```
#include "TimeDataspace.h"

TTErr TimeDataspace::test(TTValue& returnedInfo)
{
 int errorCount = 0
 int testAssertionCount = 0;
 TTValue v, expected;

// Create dataspace object
 TTObjectPtr myDataspace = NULL;
 TTErr err = TTObjectInstantiate(TT("dataspace")
     , (TTObjectPtr*)&myDataspace, kTTValNONE);

// Setup test condition
 myDataspace->setAttributeValue(TT("dataspace"),
     TT("time"));
 myDataspace->setAttributeValue(TT("inputUnit"),
     TT("midi"));
 myDataspace->setAttributeValue(TT("outputUnit")
     , TT("Hz"));
 v = 69.0;
```

```
18  expected = 440.0;
19
20  // Action
21  myDataspace->sendMessage(TT("convert"), v, v);
22
23  // Compare actual result with expected result
24  TTTestAssertion("MIDI note 69 to Hz",
        TTTestFloatEquivalence(TTFloat64(v),
        TTFloat64(expected)), testAssertionCount,
        errorCount);
25
26  // Additional tests can follow here ...
27
28  // Report results
29  return TTTestFinish(testAssertionCount,
        errorCount, returnedInfo);
30  }
```

These test methods can be called from many C++ friendly environments. We use Ruby to handle error counting, benchmarking, and logging of the testing results. This is particularly useful when doing debugging because one can run a test very fast from the command line without the need to start Max. These commands can be saved and recalled as simple Ruby scripts:

```
1  #!/usr/bin/ruby
2
3  require 'Jamoma'
4
5  puts "TESTING TIME DATASPACE"
6  o = TTObject.new "dataspace.time"
7  o.send "test"
```

Below is an excerpt of the outcome when running the script from the command line:

```
1  TESTING TIME DATASPACE
2          PASS -- MIDI note 69 to second
3          PASS -- Cent value 5700 to second
4          PASS -- Cent value 6900 to second
5
6          (...snip...)
7
8
9                  Number of assertions: 29
10                 Number of failed assertions: 0
```

3.2 Integration testing in Max

We developed a system to test the Jamoma externals within Max. This testing system consists of a couple of Max abstractions and a so called test harness. To ensure basic functional assurance, all abstractions in the testing suite are deliberately built from native Max objects rather than from potentially less reliable 3rd party externals.

3.2.1 A simple test example

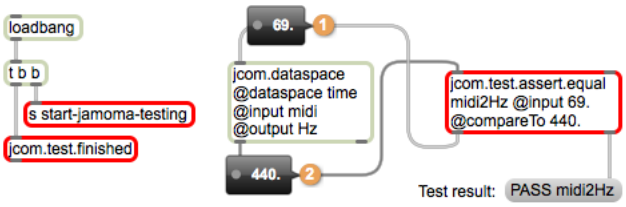

Figure 2. A simple example for testing a Max object

Figure 2 shows a simple example of an integration test for our `jcom.dataspace` external, which converts values across a variety of units in different contexts e.g., in the context of gain units, or time units. This example tests the conversion of a midi pitch into a frequency value (similar to Cycling'74's `mtof` function). The `jcom.test.assert.equal` provides the main test functionalities: sending data to a connected external or subpatch under test, receiving data from it, and comparing them. In this example, we want to test if a midi note 69 is correctly converted to 440 Hz. Therefore we set up `jcom.test.assert.equal` with the attribues `@input 69.` and `@comparedTo 440.0`. There are a few additional attributes, e.g, `@issue` provides an URL to one or more issues logged at our bugtracker web site [E]. When the test patch is loaded, `jcom.test.assert.equal` receives the execute message `start-jamoma-testing` and sends 69.0 to the connected code under test (see ① in Figure 2) which conversion result ② gets returned to `jcom.test.assert.equal`. The test abstraction then compares the received value with the expected value 440.0. If these numbers are equal, the test passes and the test results are reported back as:

```
PASS midi2Hz
```

If the test fails, the reason for failing is reported too e.g., in case the received value would be 12.2:

```
FAIL midi2Hz RECEIVED: 12.2 EXPECTED: 440.0
```

With an additional `@issue` attribute, the output of a failed test would show the URL to the bugtracker, indicating that this bug is known, as well as providing convenient access to the project resources for further details on the bug:

```
FAIL midi2Hz RECEIVED: 12.2 EXPECTED: 440.0 --
     http://redmine.jamoma.org/issues/1000
```

When all assertions in a test patcher have been processed, the `jcom.test.finished` abstraction declares the end of all tests and closes the patcher automatically. All incomplete assertions receive a timeout signal and are considered as failed.

For evaluating different aspects of the code under test, there can be multiple of such tests within one testing patch e.g., for testing different input datatypes or testing specific corner cases (e.g., testing undefined input data).

3.2.2 Testing DSP processes

For testing Audio DSP processes we have started to develop parametric tests for audio objects. Figure 3 exemplifies a test for the panpot object `jcom.panorama~`. Here, we test if the external computes a hard-left panning correctly. Similar to the previous section, the core of the test is again `jcom.test.assert.equal`. As a test signal for the code under test, we create a simple signal with the value of 1.0 using `sig~ 1.0`. When the panpot is setup via the control message `position -1.0` for performing a hard-left panning (see ① in Figure 3), we expect that the code under test will output the test signal only on the leftmost channel, whereas the right channel will be just zeros. The `snapshot~` object takes a probe of the two output signal vectors (② and ③) and `zl join` combines these two probes to a list which is then returned (④) to our testing abstraction. Besides the aforementioned attributes `@input`, and `@comparedTo`, the `@tolerance` attribute is used here

to determine a tolerance region in which the returned values can differ with respect to the expected values. This is necessary due to system-dependent rounding errors in floating point numbers as described in Section 2.2.1, but is also useful for testing features related to probabilities and stochastical processes.

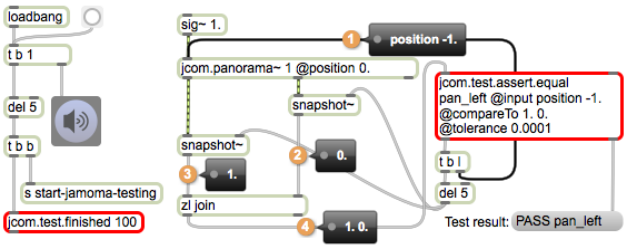

Figure 3. Testing the DSP object `jcom.panorama~`

3.2.3 Automating the tests - the test harness

For an automated execution of a larger number of tests, we implemented a so-called *test harnesses*. Our test harness (the Jamoma testrunner) performs the following tasks:

1. Loading and initializing Max
2. Gathering all tests across Jamoma subprojects
3. Consecutive execution of tests
4. Collecting test results from individual tests
5. Tracking test progress
6. Writing results to log files

The testrunner is implemented in Ruby [14] primarily to take advantage of its string parsing features. It is executed from the command line and is fully automated. At the beginning during Max's initialization process, a bidirectional OSC communication between Max and Ruby is established by loading the `jcom.test.manager` abstraction. This abstraction executes commands received from the testrunner, e.g. loading a specific test patch, and returns the test results over OSC. When all tests are executed, log files are created, providing an overview of all passed and failed test results.

The `jcom.test.manager` is also useful when manually running and monitoring one or more tests from Max itself, for instance in order to develop new tests or study the outcome of a test in further details, e.g. by loading Max Runtime from the Xcode debugger.

At the moment, the testrunner executes automatically 35 test patches with more than 400 test assertions within a few minutes. These tests have been developed as part of the process of implementing new features, according to a test-driven development approach, or in the process of debugging and resolving issues reported by Jamoma users. So far the tests are primarily addressing the core functionalities of Jamoma modules, and in particular the externals that provides the infrastructure of the modules.

4. DISCUSSION AND FUTURE WORK

In this paper we discussed the importance and benefits of automated testing systems within the computer music community. Especially when employing or re-using source code and patches on different music programming platforms, or using different operating systems and compilers, testing is crucial. We exemplify how unit tests can be integrated in the C++ code and how integration tests within Max can be executed for DSP and non-DSP patches or objects. Using a Ruby test harness we are able to regularly, efficiently, and effortlessly execute a large number of tests.

Future work includes further development of testing structures for DSP processing as well as additional tests of DSP functionalities using a variety of audio signal features. Currently, our tests use time-domain and signal energy features, but spectral-based features are often required for more comprehensive tests. When comparing longer signals, small errors e.g. in the coefficients of an IIR filter can lead to artificial delays, so comparing windowed RMS difference (or similar metrics) may perform better than a sample-by-sample check. Further, more performance tests are necessary with regards to DSP optimization and parallel computing efforts.

Within the Jamoma team, testing has become an essential tool for our development and maintenance efforts, and plays a crucial role in ensuring that Jamoma works with Max 6 as well as Max 5 for both Windows and the Mac. The tests continuously catch issues introduced in development, and it is often less important exactly what bugs are caught than how fast they are caught. It is our experience that systematic testing keeps the code flexible, maintainable, and reusable, improves confidence in the code and hence encourages bolder development cycles.

Some functionalities are easier to test than others. For instance, the testing of user-interface objects or non-linear audio processes pose challenges. By using automated tests, simple testing can be handled by the computer while leaving more time for manual testing of complicated issues.

We encourage readers to consider developing automatic testing strategies for their own computer music environments, toolboxes and artistic or research projects. The time investment will pay off sooner than one may expect. We also welcome suggestions and additions to the Jamoma testing suite which is hosted at [G] and coordinated via [H]. Because our Max testing system is free of 3rd-party dependencies, this system can be easily adopted to other non-Jamoma projects. A tutorial video illustrating how to use the testing suite can be watched at the Jamoma Vimeo channel at [I].

Acknowledgments

The initial unit test infrastructure was developed at a workshop hosted by BEK in the winter of 2011. Nils Peters is supported by the German Academic Exchange Service (DAAD).

References and Web Resources

[1] M. Newman, "Software errors cost US economy $59.5 billion annually," National Institute of Standards and Technology (NIST), Tech. Rep. 10, 2002.

[2] T. Place, T. Lossius, and N. Peters, "A flexible and dynamic C++ framework and library for digital audio signal processing," in *Proc. of the International Computer Music Conference*, New York, US, 2010, pp. 157–164.

[3] ——, "The Jamoma audio graph layer," in *Proc. of the 13th Int'l Conference on Digital Audio Effects*, Graz, Austria, 2010.

[4] T. Place and T. Lossius, "Jamoma: A modular standard for structuring patches in Max," in *Proc. of the 2006 International Computer Music Conference*, New Orleans, US, 2006, pp. 143–146.

[5] G. J. Myers, *The Art of Software Testing. 2nd edition.* John Wiley & Sons, Inc., 2005.

[6] R. C. Martin, *Clean code. A handbook of Agile software craftsmanship*. Prentice Hall, 2009.

[7] R. Cabot, "Fundamentals of modern audio measurement," *J. Audio Eng. Soc.*, vol. 47, no. 9, pp. 738–762, 1999.

[8] M. E. Takanen, "Automated system level testing of a software audio platform," Master's thesis, Helsinki University of Technology, 2005.

[9] D. A. Roberts, "High speed automated test set," in *34th AES Convention, Preprint 573*, 1968.

[10] A. Chaudhary, "Automated testing of open-source music software with open sound world and open-sound control," in *Proc. of International Computer Music Conference*, Barcelona, Spain, 2005.

[11] Y. Orlarey, D. Fober, and S. Letz, "An algebra for block diagram languages," in *Proc. of the International Computer Music Conference*, Gothenburg, Sweden, 2002, pp. 542–547.

[12] J. Smith III, *Audio Signal Processing in Faust*, http://ccrma.stanford.edu/~jos/aspf/aspf.pdf ed., Section 4.

[13] T. Place, T. Lossius, A. R. Jensenius, N. Peters, and P. Baltazar, "Addressing Classes by Differentiating Values and Properties in OSC," in *Proc. of the Int'l Conference on New Interfaces for Musical Expression*, Genova, Italy, 2008, pp. 181–184.

[14] S. Ruby, D. Thomas, and D. H. Hansson, *Agile Web Development with Rails.*, 4th, Ed. The Pragmatic Programmers LLC, 2011.

[A] `http://jamoma.org`
[B] `http://guides.rubyonrails.org/testing.html`
[C] `https://docs.djangoproject.com/en/1.4/topics/testing`
[D] `https://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/auval.1.html`
[E] `http://redmine.jamoma.org/projects/jamoma/issues`
[F] `http://ap.com`
[G] `http://github.com/jamoma/JamomaTest`
[H] `http://redmine.jamoma.org/projects/test`
[I] `http://vimeo.com/channels/jamoma/40776253`

All quoted web resources were verified on May 21, 2012.