An Automated Testing Suite for Computer Music Environments

Nils Peters
ICSI, CNMAT UC Berkeley

Trond Lossius
BEK - Bergen Center for Electronic Arts

Timothy Place
Electrotap, 74 Objects LLC

SMC 2012
9th Sound and Music Computing Conference
11–14th July 2012, Aalborg University Copenhagen

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, initially developed for the Jamoma framework and Max. The testing suite works with Max 5 and Max 6. It is independent from any 3rd-party objects, and can be used with non-Jamoma patches as well.

1. Testing in Sound & Music Computing

Stability and reliability is a general and important concern in all development and use of software. A systematic approach to testing is part of contemporary programming practice, making extensive use of solutions for running automated tests on a regular basis.

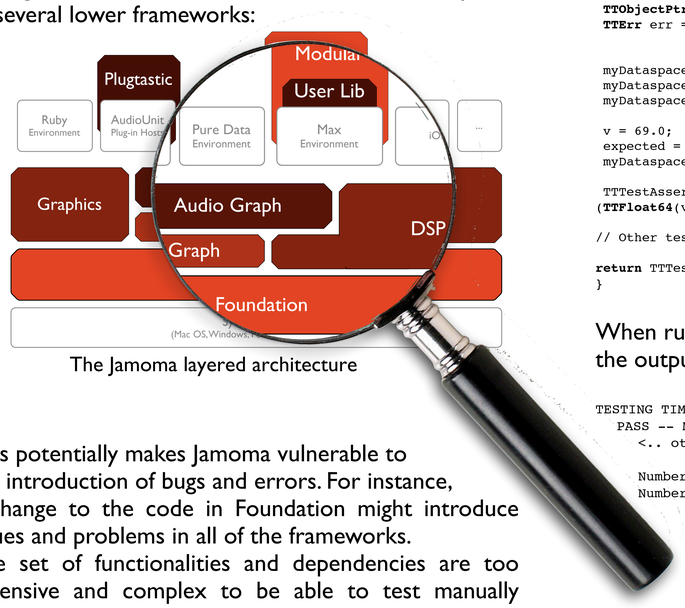
In the sound and music computing community adoption of systematic approaches to testing remain less wide-spread. To artists and musicians working with real-time media processing environments, programming is an integral part of their artistic practice. Their patches can be considered software programs, and they also become critical and integrated parts of the 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.

2. Importance of Testing for Jamoma

Jamoma is a real-time interactive media processing platform structured as a layered architecture of several frameworks, providing a comprehensive infrastructure for creating computer music systems. Jamoma is available for Windows and Mac OS with a BSD open source license. It 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.

Jamoma has a mature, well-established codebase where the higher-level frameworks, such as Modular, depends on several lower frameworks:



The Jamoma layered architecture

This potentially makes Jamoma vulnerable to the introduction of bugs and errors. For instance, a change to the code in Foundation might introduce issues and problems in all of the frameworks.
The set of functionalities and dependencies are too extensive and complex to be able to test manually whenever code is being altered. Instead a structured solution for automated testing has been developed. 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.

Most of Jamoma's Max externals are implemented as generic C++ units which are made available to Max by using a generalized wrapper function.

The C++ functionalities are validated using unit tests, while testing of the Max externals are performed as integration tests.

3. Unit Testing in C++ and Ruby

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. Unit tests can run very fast from the command line without the need to start Max by means of simple Ruby scripts.

```
#include "TimeDataspace.h"
TTErr TimeDataspace::test(TTValue& returnedInfo)
{
 int errorCount = 0
 int testAssertionCount = 0;
 TTValue v, expected;

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

 myDataspace->setAttributeValue(TT("dataspace"), TT("time"));
 myDataspace->setAttributeValue(TT("inputUnit"), TT("midi"));
 myDataspace->setAttributeValue(TT("outputUnit"), TT("Hz"));

 v = 69.0;
 expected = 440.0;
 myDataspace->sendMessage(TT("convert"), v, v);

 TTTestAssertion("MIDI note 69 to Hz", TTTestFloatEquivalence
(TTFloat64(v), TTFloat64(expected)), testAssertionCount, errorCount);

// Other tests can follow here ...

return TTTestFinish(testAssertionCount, errorCount, returnedInfo);
}
```

When running the ruby script from the command line, the output looks like

```
TESTING TIME DATASPACE
   PASS -- MIDI note 69 to second
     <.. other test are here>

     Number of assertions: 29
     Number of failed assertions: 0
```

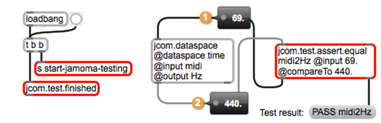
5. Test Automations - the Test Harness

For an automated execution of a larger number of tests, we implemented a so-called test harnesses performing the following tasks:

1. Loading and initializing Max as the testing environment
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

4. Integration Testing in Max

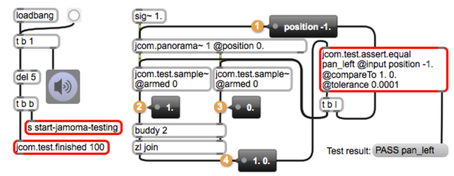
The testing system consists of a couple of Max abstractions to test Jamoma externals within Max.
This is a simple example of an integration test for our jcom.dataspace external, which converts values across a variety of units:



The `jcom.test.assert.equal` abstraction provides the main test functionalities: sending data (`@input`) to a connected external or subpatch under test, receiving data from it , and comparing with the expected result (`@compareTo`).

There can be multiple tests within one testing patch e.g., for testing different input datatypes. When all assertions in the test patch are processed, `jcom.test.finished` declares the end of all tests and closes the patcher automatically. All incomplete assertions receive a timeout signal and are considered as failed.

Integration Testing of Audio Processes



For DSP testing we have started to develop parametric tests for audio objects. The `@tolerance` attribute is used to determine a tolerance region in which the returned values can differ with respect to the expected values. `jcom.test.sample~` grabs audio samples for testing.

Conclusion & Future Work

Testing has become an essential tool for developing and maintaining of Jamoma, ensuring stability with Max 6 and Max 5 for both Windows and the Mac. In our experience systematic testing keeps the code flexible, maintainable, and reusable, improves confidence in the code and hence encourages faster development cycles. More than 600 test assertions have been created so far.
Future work includes improving the test structure for DSP processing and the integration of more audio signal features for parametric DSP testing.

www.jamoma.org
www.vimeo.com/channels/jamoma