Keeping Real-Time Electronic Music Alive

Introduction by Philippe Manoury
Proposal by Eric Lindemann and Miller Puckette

August 20, 2020

Introduction: How To Save Real-Time Music?

Eric Lindemann, Miller Puckette and I have recently had several talks about the preservation of real-time electronic music. I told them I was worried about the perpetual replacement of technological tools which put a lot of existing compositions in a very unstable position. Composers want to create a repertoire and to ensure a continuity over time, over which we want to be sure than the pieces will be performed in the same conditions. This problem is nothing new and many institutions, such as IRCAM, have made serious efforts to preserve this musical legacy. But is that enough? Obviously not. The big issue is our dependency on private companies, which we can't be sure will continue to be active in 20 years. Many of my works have been written for real time electronic music since the end the 70s, and today some of them are no longer playable. Hopefully the knowledge needed to make a piano is shared over the world and not the property of Steinway or Yamaha. But musical software, essential to perform a large repertory, is the property of a private company, which has no incentive to share its knowledge with anyone. So I asked Eric and Miller if they have ideas about how to save this situation in a solid way. This is the topic of their paper (see below) which I am happy to introduce in showing a composer's point of view.

The birth of real-time

In the early 80s, I remember a session at IRCAM, which was revelatory to me, with Pierre Boulez, Giuseppe di Guigno[1], Barry Vercoe[2], Laurence Beauregard[3], Thierry Lancino[4], and 2 or 3 technicians. It was demonstrated that a computer could follow the playing of the flutist, even in a very high speed[5]. The connection between acoustic instruments and electronic music was born. For this purpose, the electronics had to run in real time. A short time after this session I met Miller Puckette, who came to Paris for the summer. We worked closely all that time on what it has become *Jupiter*. At that time the existing library of real-time electronic tools was close to nothing: frequency shifters, ring modulators, harmonizers, delays, spatializers, reverberators and a few programs of electronic synthesis and that was it. So Miller developed a rather significant number of new tools to allow me to compose in a more sophisticated way,[6] the most important of which was the score follower. *Jupiter* premiered in February 1987[7], and just after that, Miller and I worked on *Pluton* for piano and real-time electronics which premiered, in its first version, in Avignon in July 1988[8].

---

[1] *Inventor of the 4X. The most famous piece written for this system is "Repons" by Pierre Boulez.*
[2] *Composer and researcher at MIT*
[3] *Flutist at Ensemble Intercontemporain*
[4] *Composer, at that time musical assistant at IRCAM*
[5] *The system was following the fingerings of the flute and not directly the sounds themselves.*
[6] *Synthesis by filtering a bank of oscillators, additive synthesis, rhythmic interpolations…*
[7] *Pierre-André Valade premiered it in Paris.*
[8] *The pianist was Ichiro Nodaïra.*

For this occasion, Miller had developed a new software program, *Patcher*, for resolving some difficult problems we encountered during the composition of *Jupiter*. In *Pluton*, the interaction between the piano and the computer was much more developed than in *Jupiter*. The piano was mechanically connected with the computer via MIDI[9] and acoustically by different microphones. The sounds were produced by the 4X, and the software *Patcher*, which would quickly be renamed *Max* (a tribute to Max Matthews), was collecting the mechanical information from the piano, and sending control information to the 4X.
The 4X was at this time the most powerful real time system. It looked like a big wardrobe. It was manufactured by a company, SOGITEC, as part of a system for training pilots for Aerospace and Defense systems. Musicians could only access 5 or 6 4X machines in the world and those were quite complicated to transport. But, over the years, it looked like an elegant old man who didn't run fast enough to survive in today's world, and it probably was not the main priority of the SOGITEC to invest in contemporary music research.

The second generation: ISPW

Pierre Boulez decided to initiate a new project to build a musical workstation, which became known as the ISPW, and for that Eric Lindemann was hired. The situation was better for several reasons. First, the ISPW was more powerful than the 4X; second, both the ISPW and its host computer (a NeXT) were commercialized and we could have many more machines than before; and last but not least, the machine fit into a 1-foot cube. The new capabilities of this new system were amazing. One of them was the possibility to calculate and analyze the sounds with only a microphone (the 4X needed a MIDI connection to do it). Immediately, I decided to write *En écho* for soprano[10] (it is impossible to connect a singer with a computer other than with a microphone!) because the ISPW was able to analyze the pitches and the formants of the voice in real-time. We also made a new version of *Jupiter*, replacing the MIDI cables by an audio analyser. Another advancement was the extension of Max to the domain of digital signal processing. Max wasn't only used to send information to an audio processor; it became the audio processor itself. Max/Msp was born.
I do not want to go further in this early history, but I want to point out some important aspects of the real time electronic music.

- The sounds are not pre-recorded and prepared for fixed-media presentation. For that reason I was frequently criticized by some colleagues: the sound quality was good enough, as it is in a studio, they said. I was always surprised to find that the world of performers and improvisers was much more enthusiastic than that of composers. Composers often want to notate exactly how the electronics should sound. This is a very difficult question: first, because traditional music notation is often unable to notate the sound characteristics of electronic music. Notation in that case is a guide for the interpreter to follow the electronic part[11]. Second, because it happens frequently, in real time processing, that we don't know exactly what will come out of the speakers. One of the main advantages of real-time composition is having an interaction in between the musical processes and the interpretation. So there is an element of indeterminism at the heart of real-time music. Since that time, the

[9] *The Gulbransen system was placed under the keyboard for measuring the velocity of each touch.*
[10] En écho *was premiered in IRCAM by Donatienne Michel-Dansac in 1993.*
[11] Kontakte, *by Stockhausen, is one of the most impressive examples of this.*

situation has greatly changed and, hopefully, more and more composer are nowadays working in this field.

- The pieces are highly connected with specific systems (4X, ISPW, Max, Mac, PC…) and, as I showed before, since the beginning, we had to face the problem of reproducibility and viability over the time. Today this problem is a crucial one, as we will see below.

- The more deeply a piece interacts with a given system, the less likely it is to survive. For example, today it is very difficult to perform *Pluton* for piano. The patches in Max/Msp or Pd are available, but the piano-to-MIDI interfaces are no longer manufactured and we are forced to use some old existing ones. Until when? There are attempts to run the score follower, without MIDI, by audio signal analysis[12], but until now, I have never had the chance to listen to my piece in a concert with this new system.

- In the beginning, real-time systems were mainly focused on audio processing of acoustic instruments. *Répons*, by Boulez, is one of the most famous examples of that, where the soloists are transformed in two ways: by frequency shifters (transformation of the sounds in real time) and delays (repetition of figures played by the soloists).That means that the electronics often come "after" - even if the "after" is fast enough be heard as a "during" - but never "before" the instrumental part. In other words, it was very rare to have some autonomous musical sounds superimposed on the instrumental ones[13]. I had this in mind when, with Miller Puckette, we developed rhythmic interpolations and Markov chains systems in *Jupiter*, *Pluton*, *Neptune* and *En écho*. Since that time the situation has changed. Real-time music, nowadays, includes real-time composition.

- Many pieces that are called "real time music" are, in fact, a sort of new tape music. Sounds are prepared, recorded and played back on cue, but there is no possibility to include interpretation and variability in the electronic sound. Only the timing of playback is determined in real time, not the sounds or the musical structures themselves.

Life after the beginnings

Since this period (the mid-90s), the use of the Mac and the PC have had an amazing impact on musical creation. IRCAM has produced some 800 pieces since its beginning, and several other studios around the world are working intensively. The situation which I described before, when we had only 5 or 6 computers to run our pieces, has evolved into one in which millions of personal computers are used in real-time music production. This is, of course, a

[12] *A new version is written with* Antescofo.

[13] *If we compare* Mantra *with* Kontakte *by Stockhausen, we have a very clear vision of this dichotomy. In* Mantra (1970), *one of the earliest real-time pieces, the electronic part is based only on the transformations of the pianos. There are no real compositional structures for the electronic music. Of course, at that time when computers were not able yet to produce sounds, Stockhausen had no way to do more. In* Kontakte (1957), *because the sounds were prepared in studio, the electronic compositional structures are much more developed.*

process of democratization. Anyone can now have a studio at home. Seen from this angle, the situation seems ideal. But looking more closely, one can see many clouds on the horizon. There is a danger which could be described as follows. Composers (much more than improvisers or a performers) want to create a repertory where their pieces, written 30 years ago, could sound exactly the same today. We could hazard a comparison with the baroque music where the instruments sounded quite differently than our modern instruments. There has been a lot of historical research in that field, studying the organology and performance practices of early European music. But while the musical world changed slowly over four centuries, the technological world has changed drastically over four decades! Each time there is a new generation of computers, we need to adapt our old software to the new system. And this will never stop! Much software has disappeared and we have to learn new software that adopts new forms of programming. In other words, we are obliged to learn a new music theory each decade. The great majority of musical production, nowadays, relies on commercial software (Max/Msp, Ableton Live, Cubase, Digital Performer, Synful, ProTools…) which are not open source. Imagine that those companies disappear for some reason: the pieces written with their software disappear. The sustainability of computer-based musical works is totally dependent on the evolution of industry.

How could a musical piece disappear?

I would like to give a personal example, among others, of the instability of the situation in the real time electronic world. In 2006 I have composed an 80-minute piece for singers, choir, percussion and live electronics called *On iron*. For this project I recorded texts in ancient Greek (Heraklitos) which I used to make sound materials for the piece. The main technique I used consisted of analyzing the voiced and non-voiced parts in order to separate them. The technique used at that time was called PSOLA (for Pitch-Synchronous Overlap/Add)[14]. Since that time, IRCAM has decided to abandon PSOLA, probably because they had found a better approach to resolve this issue. One could think that we could continue to use the work that had already been done. But then came a new generation of computers and PSOLA has not been adapted to it, because it was officially abandoned. If my piece were an old-fashioned one – that is, if I had recorded all the sounds and played them on a sampler – there would have been no problem. But, for many musical reasons of synchronicity with the performers, I wanted those sounds played in real time (there are always good reasons to use real-time processing). The result is easy to guess. PSOLA having disappeared, *On iron* is now impossible to perform… until somebody completely revises the code, which will be a huge project. This the reason why many composers who have composed real-time pieces prefer now to use pre-recorded sounds, transforming a real-time piece into a tape music. Farewell real time!

[14] *Norbert Schnell et al., Synthesizing a choir in real-time using Pitch Synchronous Overlap Add, International Computer Music Conference, Berlin, 2000*

Why do we have to worry now?

In the beginning, we were dependent on a small number of prototypes (like the 4X). Now we are dependent on a large number of industrial tools. The world is much bigger, but our chance of survival is more or less the same. For this reason we have to change our paradigms. We have to begin to save what is existing, and to adopt new protocols in future musical productions. Music is a serious thing, too serious to leave its fate in the hands of industry. Composers must be aware of this: if nothing is done, we will slowly, but surely, lose the music we have composed over the past 30 years!

Philippe Manoury
Strasbourg 08-20-2020

Proposal: long-term preservation of real-time electronic music

Abstract. *We propose to develop a software maintenance framework to allow the creation of new real-time electronic music that can be maintained efficiently over time. To accomplish this, we will develop a Test Suite, based on legacy patches. The Test Suite will be used to verify that new versions of software tools, such as Max/MSP and Pure Data (Pd), continue to support these legacy patches without modification. Software tools will be instrumented to run the Test Suite automatically in batch mode, comparing outputs to pre-determined "golden" outputs, to verify correct operation. Legacy patches comprising the Test Suite will be brought up to date and made to conform to a well-documented Standard Object Library. A percentage of newly commissioned works at centers such as IRCAM, ZKM, or EMS will be added to the Test Suite to assure that it continues to provide coverage of new features as they are added to the software tools. This will involve the extra work of developing test vectors for the patches associates with these new works. The test vectors will consist of known time-tagged inputs – MIDI files, audio files, recordings of mouse-clicks and other real-time interactions – and the audio or other files corresponding to the expected or golden outputs for these patches. A software team will maintain the Test Suite and develop the necessary scripts and procedures for efficiently generating test vectors, comparing test outputs to golden outputs, automatically running the Test Suite on new versions of software tools, operating systems, computer hardware and associated hardware interfaces. The developers of software tools will be invited to participate in the definition of the testing protocols, and should be expected to use the Test Suite to verify new versions of their tools. The framework could initially be developed at a leading center such as IRCAM, but over time a consortium of centers will be assembled to lend the project the greatest possible long-term stability.*

1 Introduction

It is well known in the electronic music community that most realizations of live electronic music are ephemeral. Conservatory-trained composers, whose concern is with creating scores meant as permanent documents, often avoid real-time synthesis and sound manipulation in favor of preparing sounds for fixed-media presentation. This is a great lost opportunity both for composers and classical electronic music studios that seek to bring live electronics to the classical music stage.

Composers and institutions have attacked this problem in three ways. First, a composer can restrict the use of electronics to what can be clearly described on paper in the musical score. While this does indeed offer permanence, in this approach the composer is forced to rely on simple techniques of severely limited flexibility.

In a second approach, taken for example in IRCAM's Sidney model [Lemouton and Goldszmidt 2016], materials for electronic realizations are carefully documented and versioned, allowing for the exact recreation of the files necessary for the performance of a given piece using a specific generation of music technologies. This approach greatly eases the difficulty of maintaining IRCAM's 800-some body of electronic music realizations; but

nonetheless, any piece of the 800 that is more than about five years out of date must be revised in a process called “portage”. [15] It seems unlikely that more than a few of them can be kept playable in the long term.

Many such works were developed using programming tools such as Max/MSP, Pure Data (Pd), Supercollider, etc., that have evolved considerably over the past 35 years. The computer environment that the tools run in – operating system, CPU hardware, audio, MIDI and video interfaces and associated drivers – also continues to evolve. It is this constant evolution that leads to mismatch between older musical works, newer versions of software tools, and new versions of operating systems, computer hardware, and interface devices. This is the classic problem of backward compatibility.

A third approach to maintaining works from the past, the one proposed here, emphasizes a framework for backward compatibility for the software tools themselves. Rather than portage of older works to each new version of software tools and computer environment, we propose that the software tools adapt to the new computer environments while simultaneously maintaining backward compatibility with musical and performance materials from the past.

This approach is nothing new; it was taken for example in the Pd repertory project [Puckette 2001], GAMELAN [Vincent 2010], ASTREE [Bonardi 2015], and INTEGRA [Rudi 11]. Of these, only the first seems to be still available online, but it is notable that the five pieces included in the Pd repertory project can still be performed using the files exactly as they were originally uploaded over the years 2001-2007.

This approach to backward compatibility is also used for many mainstream computer programming languages. For example, C/C++ language compilers have evolved for 50+ years but it is still possible to compile and run a C program that was written 50 years ago using a modern C/C++ compiler running on a modern computer platform. The approach to maintaining backward compatibility relies heavily on test suites of C/C++ programs with known inputs and “golden” predefined outputs. Any new implementation of the C/C++ language must be able to compile and run the programs in these test suites, taking the given inputs and producing outputs identical to the golden outputs.

In this document we propose the Real-Time Music Preservation Project. This project will emphasize backward compatibility for software tools in a manner similar to that used for C/C++ compilers. A Test Suite will be developed based on materials from older works – e.g., Max/MSP or Pd patches. New versions of Max/MSP, Pd, etc. must be able to run the patches in the Test Suite correctly prior to final release.

In order to verify correctness, patch inputs and golden outputs will be provided as part of the Test Suite, just as with a C/C++ compiler test suite. Developing these input and outputs or “test vectors”, as they are often called, will be accomplished by recording inputs and outputs during run-throughs of the works in question.

---

[15] *This term was used by French Canadian explorers to describe the arduous task of carrying a laden canoe around an obstacle such as a waterfall. The metaphor is apt.*

Automated testing will be a critical aspect of this effort. New versions of software tools must provide the capability of running the Test Suite automatically in batch mode, without human intervention, taking time-tagged inputs and producing outputs that are automatically compared to golden test-vector outputs. This automated testing will be implemented using dedicated testing computer servers that produce detailed error reports. Only when a new version of Max/MSP or Pd produces a clean error report, will it be certified as backwards compatible. It is clear from this that the Real-Time Music Preservation Project requires cooperation and collaboration with the developers of key software tools – Max/MSP, Pd, etc.

In the beginning this will be a slow and difficult process with only a few legacy patches submitted to the Project. Over time the process will become routine and will include many important patches from the past decades. This effort will also result in more standardized software object libraries and a methodology for "future-proofing" new patch development.

A large effort will be involved in developing the test vectors – i.e., inputs and golden outputs – for older patches. Because of the intense time pressure under which productions are realized, and because composers and realizers naturally seek out recently developed, and sometimes highly experimental, tools for music production, it will not always be possible for test vectors to be developed for every new work. However, institutions with investments in these software tools should provide test vectors for a certain percentage of newly commissioned works, so that the Test Suite can grow and adapt to new features provided by the software tools. This assures that backwards compatibility can be maintained into the future.

2 Components of a Real-Time Patch-Based System

A complete system for real-time computer music typically consists of software tools such as Max/MSP and Pd, an underlying computer system such as Mac, Windows, or Linux, and hardware interface devices with accompanying device drivers. A more complete description of the components of these systems can be found in Appendix 1: Components of a Patch-Based Real-Time Computer Music System.

3 Why Can't We Just Run a 10-20 Year-Old Patch?

If, after a debut performance, we put all of the elements described above in a time capsule then, barring hardware failure, we would have a good chance that the system would work 20 years in the future. In fact, there is a digital art preservation project https://rhizome.org/art/artbase/ that uses this time capsule approach.

There is a kind of time capsule approach that could work for preserving legacy musical works. This approach would create a virtual machine for every piece, using virtualization technology from companies such as VMWare. The virtual machine would be configured with the original computer operating system and all the original version of software tools, hardware drivers, patches, etc. used in the original production of the piece. The original hardware interfaces would also need to be preserved.

We do not view this approach as particularly viable. Hardware devices are likely to become non-functional over time. In addition, virtual machines run on a particular computer platform with a particular CPU architecture. Changes in the CPU architecture may cause

malfunctions in legacy patches, even when running in a virtual machine with all the original software components.

Because the computer environment – operating system, CPU architecture, hardware interfaces and drivers – has evolved, we must use new versions of the key software tools – Max/MSP or Pd, etc. – because these new versions have been adapted to the evolving computer environment.

As a result, we see as the best path towards backwards compatibility, providing the testing framework – i.e. Test Suite with test vectors and automated testing – so that new versions of the key software tools can be tested and certified as backward compatible. Again, we model ourselves on the approach to backward compatibility provided by programming languages such as C/C++. In the context of real-time software tools such as Max/MSP or Pd, the patch is to the software tool as the program is to the C/C++ compiler. The Test Suite thus consists of a collection of patches that exercise the functionality of the software tool, and provide known inputs and golden outputs to verify the correct functioning and backwards compatibility of the software tool.

4 Portage is Still Necessary

Maintaining backward compatibility is an ongoing process. Every time a new version of a software tool is introduced, it is tested against the Test Suite to assure that it can run the legacy patches. Since this discipline has been applied to C/C++ compilers for decades, new versions of C/C++ compilers are generally able to compile and run the old programs included in a C/C++ language test suite.

This test-suite based approach to backward compatibility has generally not been used for software tools such as Max/MSP and Pd. For this reason, the software tools have often evolved in ways which make them incompatible with older patches. The task now is to make modifications to the software tools and older patches so that they are again compatible, and then use the test-suite approach to maintain that compatibility into the future.

We cannot expect to modify new versions of Max/MSP or Pd in a way which makes them compatible with all older patches going back 35 years. A compromise must be found where we bring older patches up to a standard common format that will be supported by the software tools.

We intend to build our Test Suite and generate our test vectors from legacy patches. In this process we will make necessary modifications to legacy patches so that they adhere to this common standard – e.g. they all use a common Standard Object Library. Our Test Suite is expected to effectively exercise the functionality of this Standard Object Library, and new versions of software tools are expected run the Test Suite successfully.

5 Developing the Test Suite

Test vector development – i.e., recording inputs and outputs of legacy patches – is likely the most labor-intensive part of developing the Test Suite. But we do not believe that we will need to develop test vectors for all legacy patches. We believe that a reasonable sample of

legacy patches, once modified to adhere to the Standard Object Library, will provide the necessary coverage over the functional space of the key software tools.

The automated test operation will generate error reports that identify when and where patch outputs fail to match the golden outputs when a new version of a software tool is tested. These error reports will guide software tool developers in identifying necessary code modifications. In addition, once the Standard Object Library has been reasonably defined, the error reports will be instrumental in helping to port other legacy patches to be included in the Test Suite. In fixing problems with older patches, we expect to find recurring patterns of failure. By tracking these patterns, we expect to be able to develop tools and techniques for automatically porting older patches, including patches for which no test vectors have been developed.

6 Converging on a Standard Object Library

By applying the automated test strategy to an increasing number of legacy patches and including these patches in the Test Suite, we expect to converge on a Standard Object Library that supports all of the legacy patches in the Test Suite. In this process, we will likely find that different patches contain objects that have similar but not quite identical behavior. We will try to eliminate these differences in an effort to minimize the number of Standard Object Library components.

Ultimately this process should lead to a well-defined Standard Object Library. This Standard Object Library will be supported in all future versions of the software tools. As part of the automated testing strategy, simple “unit test” patches, with accompanying test vectors, should be created with the purpose of separately testing every object in the Standard Object Library.

Once a Standard Object Library has been defined, and a list of commonly occurring problems with legacy patches has been accumulated, it may be possible to automate the conversion of older legacy patches to the common standard. This includes older patches that are not part of the Test Suite, and for which test vectors are not available.

In addition to the Test Suite that assures backward compatibility for C/C++ compilers, the C/C++ language also has a formal definition that defines exactly the syntax and semantics of the language. To a certain extent, the Standard Object Library, together with definitions about how the software tool’s real-time scheduler is supposed to function, can take on the role of a formal definition. The Standard Object Library should be well documented. Ultimately, the Standard Object Library can be defined in enough detail that a third-party developer can implement it using the documentation alone, and produce a system that successfully runs the Test Suite.

The Standard Object Library should be implementable in many different software tool packages (e.g. Max/MSP and Pd). Once implemented the Standard Object Library should be able to pass the same unit tests in all software tool environments.

Once a rigorous version of the Standard Object Library is defined it can be expanded in subsequent versions of software tools. However, previous functionality must not be

modified except under rare circumstances (such as fixing bugs), and that should be accompanied with a long period of marking such objects or object features as deprecated.

Objects in the Standard Object Library should be accompanied by identifiers other than just the name of the objects – e.g. a GUID, so that inadvertent naming of custom or legacy objects with names that conflict with Standard Object Library can be disambiguated.

6.1 Likely Incompatibilities of Legacy Patches

It is worth considering what types of incompatibilities we expect to find when porting legacy patches to the new standard. This section discusses some likely incompatibilities.

- *Older objects need to be recompiled for newer CPUs.* A patch contains objects written in C/C++ that come from the software tool object library or are custom objects written specifically for a particular project. Originally, these objects may have been compiled for a different processor than those currently in use – e.g., they were compiled for a PowerPC based Mac, but all Macs now use Intel based processors. Or they were compiled for a 32-bit processor, but all processors are now 64 bits. So, the objects need to be recompiled for the new processor. However, small incompatibilities – e.g., between 32-bit and 64-bit operation – often cause the objects to malfunction when recompiled, requiring expert debugging and modification in order to run on new hardware. This type of problem will continue in the future. For example, Apple intends to stop using Intel Processors. Future Apple machines will use internally designed processors based on an ARM core architecture.
- *Older objects may use specific hardware or driver features that are no longer available.* Objects may have been written to exploit features of a hardware device or a device driver that no longer exist or have evolved in some way. The object must then be modified for the new hardware.
- *Older objects use a slightly different software tool object Application Programming Interface (API).* The rules for how the scheduler or GUI communicate with objects may have evolved since the object was first written, requiring updates to the object's C/C++ code.
- *Buffering has changed*. The original object may have made certain assumptions about the amount or order of data that arrives on its input ports every frame. The new real-time schedular or GUI might communicate with the object in a way which violates these assumptions causing the old object to malfunction in unpredictable ways.
- *Previous software tool features are no longer available or have changed behavior*. The original object may make use of software tool features that are no longer supported.
- *The software tool off-the-shelf object library evolves.* The original patch may use objects that were part of the off-the-shelf software tool object library. Those objects may no longer exist, or their behavior may have changed.
- *Subtle differences appear in the C/C++ compiler*. There may have been subtle changes in behavior of the C/C++ compiler since the original object was written.

There are, in fact, countless ways that incompatibility can create problems. The list above describes just a few examples.

7 Software Continuous Integration and Quality Assurance

A set of techniques called “continuous integration” (CI) has been widely adopted by modern software developers to help with assuring that, as new versions of software tools are developed, they do not introduce bugs or incompatibilities with existing functions. The CI approach helps to assure that the tools remain compatible with the Test Suite. CI generally involves parallel
testing and tool maintenance:

7.1 Test Suite and Test Vectors

The Test Suite of legacy patches and unit tests includes test vectors that define a set of inputs to each patch, together with a set of expected golden outputs associated with these inputs. The inputs and outputs may take the form of MIDI files, standard audio files (.wav), etc. The inputs are expected to be comprehensive enough to exercise most or all of the functionality of the patch. These inputs might be generated by recording inputs during actual rehearsals or trial runs of a performance and by recording the corresponding outputs and human interactions (buttons presses, etc.).

It is necessary to set an initial state for a patch using a scripting process. This initial state may also include seeds for random number generators used by the software tool so that the resulting outputs are repeatable. Additions to the software tool API may be necessary to allow the scripted setting of initial state and random number generator seeds.

Test vectors for a percentage of newly commissioned works should be developed and these patches should be added to the Test Suite, so that it continues to evolve as new features are added to the software tools.

The comparison between patch outputs and golden test-vector outputs can be subtle and must account for small differences in numeric accuracy, latency differences, etc. Comparison scripts that run on the automated test and verification server must be written to take this into account.

Since patches can be quite complex, test vectors should include inputs and outputs from a selection of internal patch nodes, so that when test and verification errors occur, the source of these errors can be fairly easily located, and problems addressed.

7.2 Automated compilation, verification, and report generation

The software tools must periodically be recompiled, not only because of known changes in source code or operating system, but also against unanticipated problems. To do this the software itself must have an automated compilation process and a source code revision tracking system.

When changes to a software tool are checked into the repository the Test-Suite is automatically run on a test and verification computer server. A report is automatically generated that gives details of where and when errors occurred – i.e., mismatches between generated outputs and golden outputs. The errors are then used to guide modifications to the software tool so that it remains compatible with the Test-Suite. Software tools may need to be modified so that automated script-driven batch execution is supported.

There are a number of off-the-shelf CI software packages, both commercial and open-source. These include Jenkins, TeamCity, Bamboo, Gitlab, etc. These are generally used to set up a CI server that can continuously test new revisions of software tools against the evolving Test Suite, as well as testing new versions of operating systems and new hardware interfaces.

8 Steps to arrive at a CI implementation

Transforming hundreds of legacy patches to be compatible with the CI approach, especially developing test vectors for these patches, is a large task. It must be approached incrementally beginning with relatively manageable small steps:

8.1 Choose some works to be included in the initial Test Suite

Choose a small number of pieces from the repertory which, for simplicity, are already available in open-source realizations. The realizations should avoid using any but a small core of common functions, such as provided by Pure Data (Pd) without requiring any external libraries. Possibilities include *Dialogue de l'Ombre Double* (Boulez), *Jupiter* and *Illud etiam* (Manoury), *Noanoa* (Saariaho), and *Ecosphere* (Steiger).

8.2 Generate test vectors

For each chosen piece, we obtain instrumental recordings (these should already have been made during past rehearsals or performances of the pieces) and run the existing patch with the recorded inputs, also supplying (and recording) the necessary live inputs from computer operators or performers, such as pedals and mouse clicks. After checking that the output is correct (according to the composer or someone else familiar with each piece) we run the inputs through the software tools – Max/MSP or Pd – as a batch process, not interactively, and record the multichannel audio output. These form the golden test-vector outputs.

8.3 Run patches in batch mode

Next, we write a script that re-runs each patch, with its recorded inputs, and verifies that the output is exactly the same (to within appropriate tolerances). This script should be tested on a variety of different operating systems to verify that we can robustly repeat exactly the same performance. Pd has already been designed to allow automated batch processing. It may be necessary for a new version of Max/MSP or another software tool to implement this capability. If that is not possible then legacy patches should be ported to Pd. As mentioned, we view automated testing of new versions of software tools to be an essential component to future proofing legacy works.

8.4 Harden patches for long-term stability

We carry out whatever modifications are required to the patches and any custom objects so that they can run on a modern computer platform – e.g. MacOS 10.14.x – using a recent off-the-shelf version of the software tool, and recent off-the-shelf audio/Midi interface devices. We make a clear distinction between custom objects written in C/C++ that are used by the patches and off-the-shelf objects that are part of the Standard Object Library included with the software tool release.

Any objects that are taken from third parties are considered custom objects and must include source code allowing them to be recompiled. We replace as many of the custom objects used as possible with off-the-shelf objects included in the software tool release. The object is to minimize the number of custom, project specific, objects. We recompile any remaining custom objects using a recent release of a standard C/C++ compiler – e.g. clang/llvm.

8.5 Develop scripts for automated testing

We develop scripts to run the test vectors on the patches without human intervention. The scripts compare the patch outputs with the golden outputs and flag any discrepancies (taking into account possible variations due to changes in roundoff error). This, as well as other design for test capabilities may need to be done in collaboration with the software tool developers to permit automated scripted testing.

8.6 Set up CI server

We set up a CI server using an off-the-shelf CI package and write necessary scripts to carry out automated testing on the server using the predefined test vectors. We respond to error reports as necessary by changing patches, custom objects, or scripts, or, if necessary, modifying the software tool in collaboration with the developers.

8.7 Expand Set of Test Patches in the Test Suite

The above steps result in a first automated testing platform for patch maintenance. But it is just a starting point. Once the CI framework described above is implemented, the set of test patches included in the Test Suite can be gradually expanded. This is likely an effort that will take many years. As new legacy patches are added to the test set and new test vectors developed, it is likely that new custom objects will become candidates for addition to the Standard Object Library. As the effort continues, a project manager should be appointed to oversee the continuously running tests and the necessary troubleshooting and repairs.

While the developers of software tools – Max/MSP, Pd, etc. – must agree to integrate the Test Suite and automated testing in their development process, it is likely that institutions such as GRAME, IRCAM, ZKM, or EMS will continue to maintain and expand the Test Suite.

9 Documentation

All aspects of the process must be thoroughly documented. This includes documentation for the Standard Object Library, the Real-Time Schedular, the Test Suite, scripts associated with automated testing, scripts for converting old patches to the Standard Object Library format, the CI server, and the procedure for generating test-vectors.

As expertise in writing compliant, testable patches with accompanying test vectors is acquired it will also be desirable to develop a “Best Practices for Patch Writing” document to help patch developers create easily maintainable patches that have few dependencies on the underlying computing environment and leverage the Standard Object Library as much as possible.

10 Opening up to wider user communities

It is likely that an institution such as IRCAM will need to take the lead in establishing the basic workflow and methodology for the Real-Time Music Preservation Project in collaboration with the software tool developers. Once the CI framework is well established and the system documented an effort should be made to extend the CI effort to a wider community of patch developers. This should include not just musical efforts but also performance art, museum installations, etc. It should also involve other institutions and individuals that are heavy users of real-time patch-based software tool systems. They should be invited to contribute to the Test Suite.

The more global acceptance there is of the Standard Object Library, test vector development, and verification of new versions of software tools against the Test Suite the better. The goal should be to introduce a culture of longevity to software developers who have all too often offered us disposable, throw-away software tools, and to artists who have been content to rely on them.

11 Open Source (Pd) – Open Testing Examples

Pd is an open-source project. This was part of a conscious effort on the part of Miller Puckette to provide an environment that could withstand the test of time. As private companies and individual developers come and go, open-source projects retain a large degree of robustness against future changes to computer platforms because anyone can download the source code and recompile them for a new computer platform. IRCAM and Max/MSP developers as well as developers of other software tools, should also consider this path. The Test Suite should be made available so that new implementations of software tools can be proven to be backward compatible with the large number of legacy patches throughout the world.

12 Appendix 1: Components of a Patch-Based Real-Time Computer Music System

12.1 Object Library

Software tools such as Max/MSP and Pd are systems for connecting objects. These range from simple arithmetic operators to more complex objects such as oscillators, sound effects processors, table look-up objects, score followers, etc. These objects have input and output ports that are interconnected using virtual wires to form an object network – the patch. The patch is generally constructed in a hierarchical fashion using pages containing references to sub-patches that in turn contain references to sub-sub-patches, etc. Ultimately at the leaves of this tree structure are objects taken from the basic software tool primitive object library. The objects in the library are generally created using a lower level programming language – usually C/C++. The object library is part of a software tool release. Sometimes additional custom objects, written in C/C++ as part of a particular project, are used by a patch. Patches may also use objects from third parties downloaded from the internet or other sources.

1.1 Real-time Scheduler

The real-time execution of software tool patches occurs once per processing frame, where a frame consists of a number of audio samples – e.g. 64 or 256 samples. Every frame each object reads the data on its input ports and generates data on its output ports. The input

port data for an object generally corresponds to output data from some other objects. So, it is important that, during each frame, objects X and Y that generate data needed by object Z, are executed prior to object Z. The real-time scheduler component of the software tool system is responsible for determining the correct order of execution of the objects and, on every frame, calling each object in the determined execution order.

12.2 Graphical User Interface

Software tool patches are usually created using a graphical user interface (GUI). The GUI allows users to place objects on a page and to connect them with virtual wires. When the Start button is pressed the scheduler runs the object network in real-time. While running the network the user can interact with the network by pressing buttons, adjusting sliders, and performing other actions to change parameters exposed by the object network.

12.3 Underlying Computer System

Max/MSP (originally simply Max) was created in 1986 as a MIDI event processing system to control IRCAM's 4x signal processor. Later (1988-1991) the IRCAM Signal Processing Workstation (ISPW) was designed and Max/MSP was adapted to allow patches involving both MIDI event processing and signal processing objects to run on a unified hardware platform. By the late 1990s, off-the-shelf computers had become powerful enough that special hardware was no longer needed. Today quite complicated Max/MSP patches can run on modest mid-tier laptop computers. In parallel, computer operating systems (MacOS, Linux, Windows) have evolved to allow better real-time processing with minimal latency essential for musical performance. Still later Miller Puckette, the original author of Max, developed Pd, which, among other new and interesting features, serves as an open-source alternative to Max/MSP.

12.4 Hardware Devices and Drivers

Originally institutions such as IRCAM built their own audio input/output devices. This included audio I/O as well as special devices such as the MIDI flute used for the original productions of *Jupiter* by Philippe Manoury. Today most I/O for audio and video is done using off-the-shelf components: multi-channel audio devices, standard MIDI input and output devices, video cameras and monitor outputs, motion capture devices, etc. As these devices have become more ubiquitous and inexpensive the software drivers that interface to them have become more standardized with new, feature-rich application programming interfaces (API). Often these device APIs are provided by the operating systems, and hardware manufacturers simply comply with these standards.

References

[Bonardi 2015] Alain Bonardi. "Rejouer une oeuvre avec l'électronique temps réel: enjeux informatiques et musicologiques." *Genèses musicales*, Presses de l'Université Paris-Sorbonne, pp.239-254, 2015, Archives Ouvertes: hal-01242545

[Puckette 2001] Puckette, Miller. "New public-domain realizations of standard pieces for instruments and live electronics." in Proceedings of the ICMC.

[Rudi 2011] Joran Rudi and James Bullock, "The Integra Project." in Proceedings of the Electroacoustic Music Studies Conference, New York.

[Vincent 2010] Antoine Vincent, Alain Bonardi, and Jérôme Barthélemy. Pourrons-nous préserver la musique avec dispositif électronique? *Sciences humaines et patrimoine numérique*, Nov 2010. Archives Ouvertes: hal-01156710

About the authors

Eric Lindemann is the designer of IRCAM's ISPW, the first fully programmable real-time digital music workstation.

Philippe Manoury has written real-time live electronic music since the 1970s, including *Jupiter*, the first piece to use real-time score following.

Miller Puckette is the original author of the Max/MSP and Pure Data software environments.