# PLAYING WITH TIME - MANIPULATION OF TIME AND RATE IN A MULTI-RATE SIGNAL PROCESSING PIPELINE

*Georg Essl*

Computer Science & Engineering and Music
University of Michigan
Ann Arbor, Michigan, USA
gessl@eecs.umich.edu

## ABSTRACT

Time is a central notion in synthesis engines, and manipulating time is an important part of structuring an instrument, a sound or a performance. We discuss how time can be treated in a flexible-rate and multi-rate dataflow engine that does not operate on a preferred rate. We describe how rates can be locally controlled, now interweaving rates can be managed. A multi-rate pipeline has benefits both for computational load as well as ease of building dataflow interactively in live performance.

## 1. INTRODUCTION

In this paper we address the manipulation of time and rate within a multi-rate signal processing pipeline. Time is of course a central notion to audio processing, as it is intrinsically a time-based medium. Many perceptual qualities change critically with time differences such as the transition of temporal to frequency hearing [16] and manipulation of time has long been recognized as a central aspect of control. Many classical techniques in digital sound synthesis are modifications of parameters over time. For example amplitude envelopes are the such time-based changes of the overall signal amplitude [6].

Not only controls are subject to timing. In fact the audio or other relevant signal data is streamed over time, usually with a constant sample rate. In fact the vast majority of sound synthesis environments use the audio rate as the single given rate for all its signal flows. This choice is quite natural if indeed the purpose of the pipeline is primarily audio processing. This dominance of the audio rate has lead to signal data and the control data to be seen as distinct types of information that vary over time. Signal data is updated at a fixed single data rate while the control signal can change whenever a control event happens, which may be at regular intervals, or in irregular patterns.

However, as already described in [9] this distinction between signal and control can be abandoned. They can be treated in some sense as different versions of a general stream of data that changes discretely over time. This automatically leads to a digital signal processing pipeline that operates on variable rates.

A major reason for removing the distinction was the goal of building a digital synthesis dataflow paradigm that is well-suited for on-the-fly live performance. Here the goal is to make changes to the pipeline easy and fast. This distinction proved to be an obstacle to rapid switching of units. Consider using a microphone as input to the amplitude of a sine oscillator. Hence the microphone signal would serve as a kind of amplitude modulator. Traditionally the microphone would be considered providing an audio signal, hence the semantics of it is that of a signal. However if we want to switch the microphone for say an accelerometer, or an interface slider, or some other traditional control input, that semantics changes. In the traditional view the new connection would be seen as control. So the performer changing that input live, would have to not only change the input but the semantics of the connectivity.

A related issue emerges when switching outputs. For example someone may want to not only process data for audio output, but also provide processed information for a visualization, and send data over the network. However visual information renders at a different rate as audio, and the need for sending networked data may depend on the goals of the particular performance and too may not be best serviced at audio rates. This immediately leads to a very related undesirable effects of having a preferred sample rate. Some possible output modality do not match, hence there is a need to translate. Yet we want to support these changes on-the-fly in live performance, hence extra steps to ensure this translation again is a burden.

Be removing that the user pay attention to these differences, rapid patching becomes possible, but it leave the requirement of the dataflow pipeline to handle these differences in rates and timings.

Here we will discuss in detail the design of such a pipeline with an eye toward manipulating time and rates and the characteristics of the pipeline design that help decide local timing and rates. And we will discuss the kinds of time and rate based processing units we suggest for such an audio processing pipeline.

## 2. RELATED WORK

The work presented here is currently in use in the mobile programming environment UrMus [7]. Audio processing engines have a long-standing history going back to its ori-

gins with Music I by Max Matthews. Ultimately multiple paradigms have emerged addressing how to allow users to generate and process music. The most dominant paradigms are text-based systems, such as CSound [3], Arctic/Nyquist [5, 4], SuperCollider [15] or ChucK [21] on the one hand, and graphical patching systems, such as Max/MSP [20] or Pure Data (Pd) [19] on the other hand.

The importance of time has been recognized for a long time and many systems described above have a wealth of mechanisms to deal with timing, rate and changes in dataflow.

However a single audio-rate is the dominant paradigms even in systems that do offer rich primitives to control time itself, such as is the case in ChucK or SuperCollider. Even though Pure Data is in principle a single-rate dataflow system, aspects of multi-rate processing can be implemented in Pd via the sub-patching mechanism [14].

The question of incorporating timeliness into multi-rate dataflow architectures, as well as a range of time-based flow patterns was addressed by Azumi and co-workers [2, 1] and they also addressed suitable visual representations for time-based patterns [10]. The use of serialization, vectorization, decimation and selection operators to offer pathways to multi-rate processing was explored in the contexts of the designs of Faust [12, 13]. Our concern here is somewhat different to these prior proposals as we discuss in the following section.

Perhaps the closest to the current work is the recent work by Norilo [18] on the Functional Reactive Paradigm invented by Nordland in the context of functional programming [17]. The key idea here is that from the call order in a functional evaluation, the local frame rate requirements can be deduced when considering *sources* (he used the word spring) are found in the functional evaluation path and their rate propagated to points of intersection and ultimately connected all the way to the rendering hardware called *sinks*. In our work we arrive at similar conclusions, however outside the functional paradigm and we arrive at a structure that is directional. In addition our introduction of directionality, the approaches also differ in the way they resolve rate conflicts. Norilo proposes a priority scheme. Here we will consider direct interventions to decide locally within the dataflow graph how rate is determined.

## 3. WHERE DOES TIMING INFORMATION COME FROM?

Here we want to addresses the question: Where should the local time advancement in a dataflow come from? Hence we want to address how to reason about timing control locally, how to understand how timing propagates, or stops propagating within the network and how it can be controlled and manipulated.

There are numerous sources of timed information. Input happens over time, or the source of input may itself be driven by a regular or irregular time-pattern. For example microphone input happens at audio rates, while accelerometer input happens at a significantly lower yet also regular rate, touch input happens irregularly. Similarly output modalities have their intrinsic timing. A network request may demand process in an irregular fashion, visual, or audio information is rendered at different regular, and in the case of visual scenes, even irregular basis.

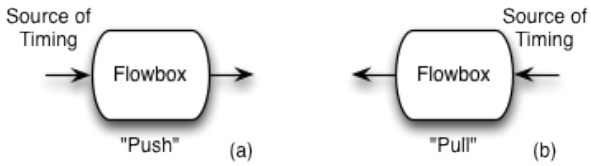


**Figure 1**. The depiction of directionality in our multi-rate dataflow. (a) A Push refers to the rate being derived from the input side. (b) A Pull refers to the rate being derived from the output side.

From this we would argue that typically time in a dataflow pipeline has the following typical sources:

1. Rates of input and output hardware connected to the data flow.

2. Irregular events induced by user actions or authored time lines.

However there are further considerations that impact computational cost. For example a long-time averaged signal needs to be updated much less frequently than a live audio signal, even if the control signal it is based on was triggered more quickly. Hence within the network there may be trade-offs in terms of local rates.

Ultimately we want as principle that the local time update, whether rate or irregular timed event dictates the computation, rather than a global pulse operating at a high sample rate.

## 4. DIRECTIONALITY AND COUPLING

A key observation is that a typical data flow processing block (we will use the word *flowbox*) can actually be updated in more than one way. Either there is a demand for new output hence requiring computation. Or there is a new input that could have a similar effect. In this paper we will call data that is brought to an input a *push* and we will call data that is being requested at an output a *pull*. A graphical interpretation of this difference is depicted in Figure 1. A push is a right-pointing connection arrow, while a pull is a left-pointing.

We observe that whether a flowbox is pushed or pulled has the effect of deciding where the update timing is coming from. If it is pushed, the output-side drives the operating rate, whereas if if is a pull, it is the input side. Hence this choice of directionality is linked to a choice in the source of timing.

There are two types of relationships between input and output of a flowbox. One is that updating one requires an

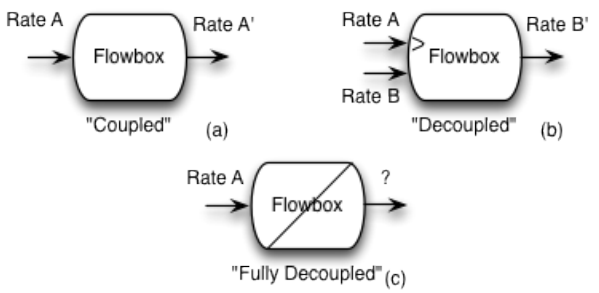


**Figure 2**. Flowboxes can have different behavior with respect to propagating timing. (a) Coupled input-output pairs propagate timing. (b) A decoupled pair will not. (c) A fully decoupled flowbox has no coupling input-output pairs.

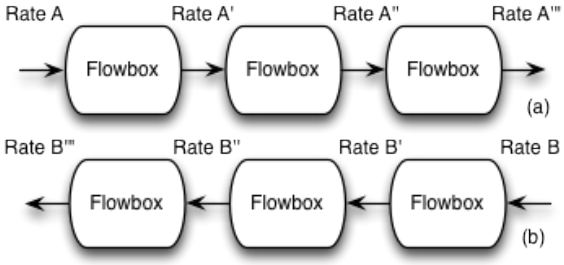


**Figure 3**. If flowboxes couple, timing information will propagate through them. In general they may be motified though often that is not the case. (a) Input-side or "push" timing information. (b) Output-side or "pull" timing information.

update of the other. We call this property *coupled*. For example standard linear time-invariant filters are coupled. A new output has to be computed when a new input arrives, or if updated on the output side, a new input has to be acquired if a new output is requested. Coupled flowboxes occur often in dataflow and they typically correspond to being fed through in a fashion that is dictated by arriving (or leaving data). Usually that means that the sample rate, or irregular timing is unchanged, though one can design coupled flowboxes that do not have this characteristic. For example a down-sampler may emit only half the samples that arrive at its input but the rate of emission is still directly linked to the rate at that input.

However, not all relationships between input and output have to be coupled. Take for example a simple gain with a gain input and a signal input and output. Clearly changes in the gain input do not require an immediate change in the signal input or output. It is sensible behavior to only apply the current gain when the signal path is being updated. We call this property of the gain input with regards to the signal output *decoupled*.

Decoupling has an important consequence for timing, as it breaks the relationship of input and output with respect to timing. Timing does not have to, but could be forced across a decoupled input-output pair. This property is central to deciding and controlling timing in various sub-flows of the dataflow network.

For example a gain is set to a certain value. After that, no change in the signal flow through the gain requires an update of the gain value. Another way to think about the difference between coupled and decoupled input-output pairs is with respect to required computation (and its cost). A coupled pair will require computation when invoked, whereas the decoupled pair may not. For the rest of the paper we will use the symbols as depicted in Figure 2 to represent the different cases. The coupled flowbox simply uses a blackbox representation. Decouple either uses semi-circles around the decoupled input or output, or if all inputs and outputs are decoupled, uses a diagonal separation line. We call this last instance *fully decoupled*.

Fully decoupled are actually also regular entities within typical dataflow processing pipelines. Consider traditional unit generators, such as a sine oscillator. It has as output a signal, and a number of input parameters that control the nature of that signal (amplitude, frequency, phase). Notice that none of these inputs has to be a signal and no update of any of them necessarily means that a new output has to be computed right away. Hence a sine oscillator is fully decoupled.

### 4.1. Consequences of Directionality and Coupling

Most generally, a flowbox may indeed be used in both direction even in a concurrent or interleaved fashion. In principle invoking a coupled input-output pair simply invokes the internal computation of the unit. However, given that a flowbox is connected to other flowboxes there is a propagating consequence to directionality. If a coupled flowbox is updated in one direction and connected to another flowbox, that flowbox is invoked with this directionality as well. As long as all flowboxes in the chain are coupled, clearly all of the flowboxes will be updated (see Figure 3.

A decoupled flowbox ends this requirement to propagate updates. Hence coupled chains of flowboxes have their timing determined at either the output, or the input of the chain (or perhaps a rate generating flowbox in the middle of the chain, to be discussed later). A decoupled flowbox can make the timing undetermined. It is easy to construct examples with undetermined timing for a subflow of a dataflow network. Consider Figure 4. We have two cascaded sine oscillators connected to an accelerometer input on one side, and the digital analogue converter (dac) for audio playback on the other side. The input of the first oscillator will be updated at the accelerometer rate, while the the output of he second oscillator is updated at audio rates. Notice however that due to the fully decoupling nature of both sine oscillators, the flow between the two has no determined timing.

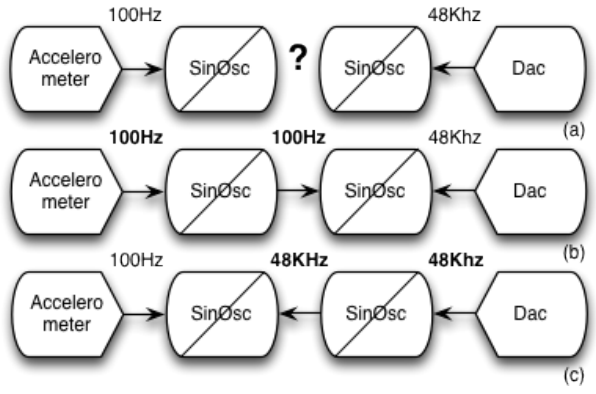


**Figure 4**. (a) Due to the decoupling nature of the sine oscillator flowbox, the timing between two connected sine oscillators is not inherently determined. (b) Choosing a push connection informs the system to take input side timing. (c) Choosing a pull connection directs the system to use output side timing.

### 4.2. Overcoming undetermined timing

In single-rate architectures, this problem of undetermined timing does not occur, because a global update rate is imposed on all parts of the dataflow network. But as discussed in the previous section, in the proposed multi-rate design, parts of the dataflow network can become isolated from timing mechanisms and hence have no determined timing.

There are numerous possible mechanisms to overcome this problem. Perhaps the most natural is to choose to propagate timing through a decoupled input-output pair. Figure 4 shows the two possible solutions in this case. If the timing is fed through from the dac, that is, the second oscillator pulls from the first, the rate on that connection and the update of the output of the first oscillator will be at audio rates. However if we push the accelerometer rate through the first sine oscillator to the input of the second, that input will now update at the input rate. Notice how these two options are not equivalent, and the pushing into an input, or pulling from an output dictate which rate is going to be used.

For this reason directionality becomes an important piece of information and we always draw the connections between input and output as arrows. In terms of timing, this describes a master-clock relationship. The tail of the arrow is the source (or master) of the timing information, and the tip of the arrow is the recipient of the timing information.

### 4.3. Controlling undetermined timing via Pumps

Aside the two options we just discussed one might want to be able to inject a timing behavior different from the input and output sources surrounding the decoupled patch. We solve this requirement by introducing a special class of flowboxes that we call *pumps*. It has a coupled input-output pair, as well as additional special inputs or outputs.

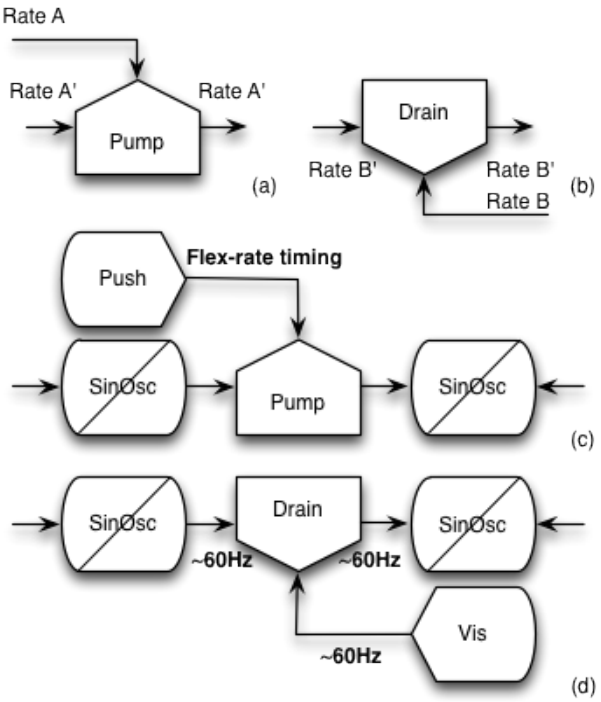


**Figure 5**. Pumps allow for rate injection in data-flows with undetermined timing. (a) A pushing pump. (b) A pulling pump or drain. (c) A variable time push event driven pump. (d) A visual update driven drain.

The idea of the pump is that the rate at those special connectors (whether pushed into it, if it is an input, or pulled from it, if it is an output) will dictate the rate of the coupled connection pair. Figure 5 shows example pumps inserted between the two sine oscillators of our example. If a pulling connection is used to drive a pump we call it a *drain*. Here the visual update rate is used to control the timing. Notice how the link between the two oscillators operates on a different timing schedule than either input or intended output. The second pump is driven by a push flowbox. This is a flowbox that can be programmed or linked to event-based user input (like touching the screen) to create timing information. Hence the update between the two sine oscillators is now determined but completely arbitrary.

One can define pumps with or without side-effects. Pumps without side effects simply use the rate of the signal at the pump input to drive timing but ignore the data. These pumping mechanisms without side-effect have in special form already appeared. For example in ChucK, a blackhole [21] is a pulling pump (drain) without side effects linked to the audio rate. In our multi-rate system this can be accomplished using a drain as seen in Figure 5 but connecting it to the dac to get audio rates.

Pumps with side-effect use this data to additionally affect the pumped signal. For example one can define a gain-pump, which is a hybrid of a gain and a pump. When the gain is set, the pump also update the coupled input-output. If both gain and input-output are connected

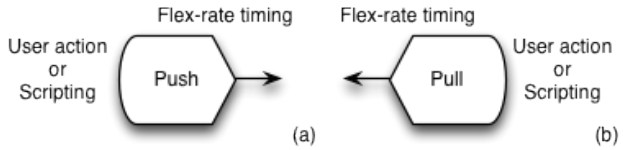


**Figure 6**. User input can provide timing information. Push (a) and Pull (b) flowboxes allow user-based, or scripted timing to be feed into the dataflow.

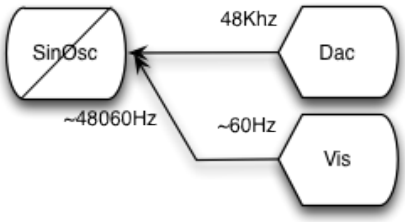


**Figure 7**. The depiction of directionality in our multi-rate dataflow. (a) A Push refers to the rate being derived from the input side. (b) A Pull refers to the rate being derived from the output side.

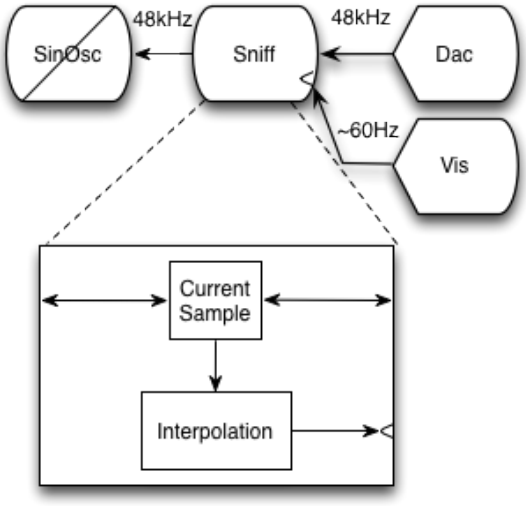


**Figure 8**. The use of a Sniff flowbox to allow a dataflow with one rate to observe another dataflow's timed stream without interfering with the timing itself.

to audio-rate data, this would be the same as a ring modulator [6] whose overall rate is driven by the timing at the gain input.

### 4.4. User, event, or script based timing

So far we have only discussed a certain set of timing sources in detail, namely rated inputs (like accelerometers) and rated outputs (like audio output). However there clearly is a need for timing that is driven more directly by the user. Say a user pushes a button on the screen as part of a drumming performance. This timing clearly should be usable within the dataflow. In order to allow this, we create a new source and a new sink, which we call *Push* and *Pull*. These are flowboxes that can have their input set programmatically as is the case for a Push (see Figure 6). That program can either be linking the push to a user action such as a button press, or some other event, or a script that generates timing. A Pull does the same except that it allows the program to read out data using programmed timing. Here too the timing can come from user action on the interface. Their behavior is similar to bangs or number events in MAX/MSP or Pd in principle [19] except that due to the directionality choice we intrinsically consider dual pairs (Push and Pull).

## 5. HANDLING MULTIPLE SOURCE OF TIMING

An immediate consequence of a multi-rate pipeline is the presence of more than one source of timing and that those sources of timing can have different patterns. For example a pipeline may want to have two sinks update the overall flow. We allow this by simply allowing multiple pull links to connect to an output. However that means that the effective time pattern is the interweaving of the time patterns of each of the sinks. In some cases this may be unproblematic, or even desirable. For example imagine that two user-driven events can progress time. Then the interweaving of those events is precisely the desired behavior. If two interweaved timing are steady rates, then the interweaved rate is very likely an irregular time pattern with an effective rate different from either of the interweaved rates. The exception is the case when both rates are the same, in which case we get a doubled regular rate.

Consider the example depicted in Figure 7. Here both the dac and the visual output are connected to a dataflow network as sinks. The dac updates at 48kHz and the visual output updates at an irregular but averaged steady rate of 60Hz, depending on CPU load. Hence with interweaving, the dataflow network is updated at an effective irregular rate of 48060Hz. Clearly this is not desirable for audio playback due to artifacts introduced by the injected updates from the visual side and it may not be the desired behavior for the visualization either. In fact what is likely the desirable outcome in such as case is that indeed one source of timing dictates the timing of the dataflow network, while the other sink only wants to read out information.

However we already have a mechanisms that in principle can address this: decoupling. Hence by taking any decoupled output of a timed dataflow that we want to observe without affecting the rate of that dataflow we can do this. However we may want to provide a decoupled flowbox with that explicit function without additional side-effects. We call this flowbox *Sniff*. It consists of a coupled input-output pair, as well as a decoupled output. The effect of this flowbox is to observe a running timed dataflow at some chosen times. Hence it acts as a resampling of one stream by another. In fact resampling in this naive fashion may not be the best choice in some situations, hence the simple Sniff serves as the simplest prototypical examples

of a general class of resampling flowboxes, that may not only observe but use a derived observation (such as interpolations) to achieve a desired resampling outcome (see Figure 8).

The dual argument also applies for inputs. Input-side timing if connected to the same input will interweave. However pragmatically we found this to be not a particular issue. Inputs tend to either be well mixed by existing decoupled flowboxes, such as gain, or terminate in a decoupled input, hence the interleaved pattern having no particularly relevant impact.

However in principle a merged input with a steady rate can be achieved by synchronizing injection of one dataflow with the other and this synchronization too may be subject to processing such as interpolation. There is a wide choice of thinkable injection functions. In fact the gain flowbox we have discussed earlier is a form of injection.

Finally one can give control to the rate choice to the dataflow network itself by providing. This can be achieved through a pair of flowboxes that allow the selection of the input or output, respectively that is currently used to derived rate. For inputs lines that are not selected to provide rate information can either be ignored, or injected. For output, those lines too can either be ignored or allow observation of the data stream.

## 6. MANIPULATING TIME AND RATES DIRECTLY

We call a flowbox *time-manipulating* if part of its effect is the change in temporal pattern between coupled input-output pairs.

A range of such effects is thinkable. An illuminating example is the *ZPuls* flowbox. ZPuls observes a data stream and emits an impulse whenever it observes a zero-crossing. More generally we call flowboxes who emit data based on conditions on the data stream *conditionals* [9]. It will, however be silent if no such zero crossing occurs. Clearly this flowbox is time-manipulating. The timing of the pulse pattern generated by it is very likely very different from the rate of the signal observed. ZPuls in fact is a very general flowbox that can be used to generate rich regular and irregular timing patterns with the pattern controlled by the wave form. The key property here is of course that an impulse and hence followup computation is only invoked when it is relevant, and this can be made arbitrarily rare. ZPuls is a prototype for a mechanism that we call *retriggering*. Not all incoming timed data propagates. Rather whether some outcome timed event is triggered is conditional on some property.

Another class of time-manipulating flowboxes have to do with downsampling. Consider downsampling by a factor of two. The simplest possible way to achieve downsampling is to simply omit every other sample from a signal. Traditionally in this is often seen as setting the omitted samples to zero. We have however to the option here to simply only send data half of the time, hence manipulate time at the same time as we reduce the actual data. A general down-sample at an arbitrary downsample rate will have to interpolate, hence leading to a general solution to down-sample to arbitrarily lower rates.

An adaptive downsampler can be build from a simple differentiator, which we call *DiffGate*. This flowbox will only emit data if there is change input-side from previous data. If the data is indeed slow-changing and over large stretches constant this flowbox will remove the constant-rate behavior that does not actually give changed data. A related flowbox *QuantGate* will only emit a new sample if the accumulated difference to the last emitted data is larger than some specifiable threshold. The effect is that only changes above a specified quanta will create a timed event. This flowbox can be used to adaptively downsample slowly changing but non-constant signals. This too is a very general flowbox that works well to adaptively downsample slow-changing effects such as amplitude envelopes.

## 7. UNIVERSALIZATION OF PROCESSING UNITS WITH TIME-VARIABILITY

Already in [9] we discussed the property of universal plugability with respect to being able to connect processing units with different input and output semantics by defining a normed data-stream and making each processing unit understand how to translate the normed data into its own suitable semantics.

Here we also get a universalization with respect to time primitives. For example consider that a drum machine is to be built, where different weights of impact should be stored in a circular buffer. However we already have flowboxes that are meant to store circular buffered data and allow its playback and rate control. This flowbox is called *Looper* and was originally designed to allow recording and looped playback of audio samples. However, in our pipeline design there is nothing privileged about the audio rate, or even a regular rate. We can hence use Looper at an arbitrary, if we want significantly slower, or irregular rate, Hence we can use the samples stored in Looper as the intensity levels of a drum and play it back at rates appropriate for a typical drum rhythm. We can even chose to make the timing irregular, to perhaps allow a swing pattern to apply, without having to define a Looper block different from the one we already have. In this sense all flowboxes that we do have now can be used in different timing situations. For example an averaging filter can be used on irregular data.

## 8. REALIZATION IN URMUS

What we have discussed here is realized in urMus, a interaction environment for commodity mobile devices [8]. However urMus hides many aspects of the rate control from the user in the default interface. For example the directionality of the flow is not visible in the visual representation of the interface (see Figure 9). This is purely

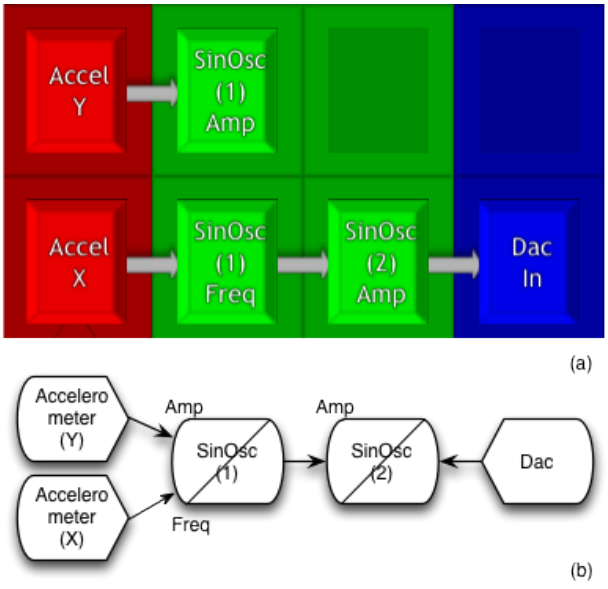


**Figure 9**. (a) An amplitude-modulation patch in urMus. Notice how the directionality of the connectivity is hidden in the representation. (b) The actual patch structure as implemented, using our symbolic notation.

a choice of representation. The actual connectivity generated from the representation does retain the directionality we described here. To allow this, urMus uses an algorithm for finding sensible sources for rate control for its network. This algorithm is based on two principles: Input should dictate rates if there is undetermined rates as input rates tend to be lower, hence less computationally expensive. The second is to propagate rates from inputs and outputs until points of decoupling.

The complete algorithm is as follows:

1. Connect from all inputs in push direction until a decouple (or an output) has been reached.
2. Connect from all outputs to all unconnected flowboxes until a decouple (or an input) has been reached.
3. For all undetermined directions, pick the input for rate control.

It is easy to see that this algorithm covers all possible cases that can be constructed by the urMus interface.

A general principle in urMus is the separation of visual representation from programming concept. Hence there is no canonical way to represent the dataflow. In fact the dataflow network could easily be reskinned to look like other familiar or new representations, whether textural or graphical.

However the choice of the current representation has to do with the requirement to allow rapid patching in a live setting. The multi-rate property of the flow is computationally efficient and allows to remove the distinction betweeen control and signal flows, but often knowledge of rates is actually not explicitly required. Notice also that individual inputs are presented as separate interaction entities (SinOsc(1) Amp and SinOsc(1) Freq are separate widgets). This too is justified to organize interactions into finger-sized elements.

### 8.1. The Lua interface

UrMus itself has a Lua layer. Lua is a fast, embeddable script language with certainly attractive properties for malleable user interface design [11]. Within the Lua language, aspects of UrMus that are otherwise separate can be programmatically related. The dataflow pipeline discussed here is exposed to lua through API functions.

Key aspects of the interface is the possibility to create new instances of flowboxes, and to change the connectivity between them. The function to create a new instance is `CreateFlowBox(prototype)`. For example

```
mySinOsc = CreateFlowBox(_G["FBSinOsc"])
```

creates a new instance of the SinOsc flowbox from the global prototype FBSinOsc. Some flowboxes are stateless and hence do not require instancing. Many hardware sources and sinks are of this type. For example we can use

```
dac = _G["FBDac"]
```

to get the global instance of the audio playback sink. In order to establish a pull link between the sin oscillator and the dac we use the method outflowbox:SetPullLink(inindex, inflowbox, outindex).

```
dac:SetPullLink(0, mySinOsc, 0)
```

Or alternatively one can use member functions of the flowbox itself.

```
dac.In:SetPull(mySinOsc.Out)
```

The moment this is executed, a sine will play at default frequency of 440 Hertz and at normed amplitude. To connect this to an accelerometer that pushes into the frequency we do the following:

```
accel = _G["FBAccel"]
accel:SetPushLink(0, mySinOsc, 0)
```

And one can of course also use the flowbox member notation.

```
accel.X:SetPush(mySinOsc.Freq)
```

To undo connections one can use `:RemovePullLink()` and `:RemovePushLink()` or `RemovePull()` and `RemovePush()` in flowbox member notation, with the same arguments that one used to create the connection. These sets of functions and methods are already sufficient to generate fully functional dataflow networks and through directionality control rates. However there are further methods provided that allow an interface to sensibly represent a flowbox in whichever form it prefers. So can a flowbox be queried for the number and labels of each input and output it offers via the methods `:NumOuts()`

`:NumIns()` `:GetOuts()` `:GetIns()`. One can find out if a flowbox can be instanciated by querying via `:IsInstantiable()`. There are also methods to query the (de-)coupling property via `:GetCouple()` and `:IsCoupled()`, and to push numbers into or pull numbers from a data network via `:Push()` and `:Pull()`.

## 9. CONCLUSIONS

In this paper we presented the design of a flexible-rate and multi-rate dataflow architecture that does not have an intrinsic master rate. The rates of local sub-flows of the dataflow network are derived from the rates of the hardware and interaction elements that are connected to it. We gave how the variable rates within the network can be determined and controlled and gave a range of mechanisms to manipulate time.

There are a number of benefits to this design. One is that it is well-behaved for live-patching [9]. Furthermore it allows subparts of the network to run at computationally relevant speeds, not the usually higher audio rates, allowing better performance specifically on the still slower mobile hardware. Finally having flexible control over time within a dataflow leads to a universalization of the use of flowboxes that is liberated from the assumption of a dominant single rate.

## 10. REFERENCES

[1] P. Arumí and X. Amatriain, "Time-triggered static schedulable dataflows for multimedia systems," in *Proceedings of SPIE*, vol. 7253, 2009, p. 72530D.

[2] P. Arumí, D. García, and X. Amatriain, "A dataflow pattern catalog for sound and music computing," in *PLoP '06: Proceedings of the 2006 conference on Pattern languages of programs*. New York, NY, USA: ACM, 2006, pp. 1–23.

[3] R. Boulanger, *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. Cambridge, MA, USA: MIT Press, 2000.

[4] R. Dannenberg, "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis," *Computer Music Journal*, vol. 21, no. 3, pp. 50–60, Fall 1997.

[5] R. Dannenberg, P. McAvinney, and D. Rubine, "Arctic: A Functional Approach to Real-Time Control," *Computer Music Journal*, vol. 10, no. 4, pp. 67–78, Winter 1986.

[6] C. Dodge and T. Jerse, *Computer Music: synthesis, composition and performance*. Macmillan Library Reference, 1997.

[7] G. Essl, "UrMus – an environment for mobile instrument design and performance," in *Proceedings of the International Computer Music Conference (ICMC)*, Stony Brooks/New York, June 1-5 2010.

[8] ——, "Urmus-an environment for mobile instrument design and performance," *Proceedings of the International Computer Music Conference*, 2010.

[9] ——, "UrSound — Live Patching of Audio and Multimedia using a Multi-Rate Normed Single-Stream Data Flow Engine," *Proceedings of the International Computer Music Conference*, 2010.

[10] D. Garcia, P. Arumi, and X. Amatrinain, "Visual prototyping of audio applications," *LINUX AUDIO*, p. 88, 2007.

[11] R. Ierusalimschy, *Programming in Lua, Second Edition*. Lua.Org, 2006.

[12] P. Jouvelot and Y. Orlarey, "Semantics for multirate faust," *New Computational Paradigms for Computer Music-Editions Delatour France*, 2009.

[13] ——, "Dependent vector types for data structuring in multirate faust," *Computer Languages, Systems & Structures*, 2011.

[14] V. Lazzarini, "Music programming systems for dsp research," Lecture Notes. Unpublished. Retrieved online on January 20, 2012. [Online]. Available: https://noppa.aalto.fi/noppa/kurssi/s-89.3580/luennot/S-89_3580_lecture_handouts.pdf

[15] J. McCartney, "Rethinking the computer music language: Supercollider," *Comput. Music J.*, vol. 26, no. 4, pp. 61–68, 2002.

[16] B. Moore, *Introduction to the Psychology of Hearing*, 4th ed. Academic Press, 1995.

[17] J. Nordlander, "Reactive objects and functional programming," Ph.D. dissertation, Chalmers University of Technology, Department of Computing Science, 1999.

[18] V. Norilo, "Introducing kronos: A novel approach to signal processing languages," in *Proceedings of the Linux Audio Conference, Frank Neumann and Victor Lazzarini, Eds., Maynooth, Ireland*, 2011, pp. 9–16.

[19] M. Puckette, "Pure data: another integrated computer music environment," in *in Proceedings, International Computer Music Conference*, 1996, pp. 37–41.

[20] ——, "Max at seventeen," *Comput. Music J.*, vol. 26, no. 4, pp. 31–43, 2002.

[21] G. Wang and P. R. Cook, "Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia," in *ACM Multimedia*, 2004, pp. 812–815.