A UNIFIED MODEL FOR AUDIO AND CONTROL SIGNALS IN PWGLSYNTH

Vesa Norilo and Mikael Laurson
Sibelius-Academy
Centre of Music and Technology

ABSTRACT
This paper examines the signal model in the current iteration of our synthesis language PWGLSynth. Some problems are identified and analyzed with a special focus on the needs of audio analysis and music information retrieval. A new signal model is proposed to address the needs of different kinds of signals within a patch, including a variety of control signals and audio signals and transitions from one kind of signal to another. The new model is based on the conceptual tools of state networks and state dependency analysis. The proposed model aims to combine the benefits of data driven and request driven models to accommodate both sparse event signals and regular stream signals.

1. INTRODUCTION
The central problem of a musical synthesis programming environment is to maintain a balance of efficient real time performance, expressiveness, elegance and ease of use. These requirements often seem to contradict. Careful design of the programming environment can help to mitigate the need for tradeoffs.

The original PWGLSynth evaluator [3] is a ugen software that features a visual representation of a signal graph [1]. It was written to guarantee a robust DSP scheduling that is well suited for tasks including physical modelling synthesis. This was accomplished by scheduling the calculations by the means of data dependency - in order to produce the synth output, the system traverses the patch upstream, resolving the dependencies of each box in turn. This signal model is often referred to 'request driven' or 'output driven'. The model has the distinguishing feature of performing computations when needed for the output, and is well suited for processing fixed time interval sample streams.

The opposite model is called 'data driven' or 'input driven'. Calculations are performed on the system input as it becomes available. This model is well suited for sparse data such as a sequence of MIDI events. The input driven model is represented by the MAX environment[6]. The bulk of PWGLSynth scheduling in the current version is output driven. Some benefits of the input driven approach are available via the use of our refresh event scheme, which can override the evaluation model for a particular connection.

The rest of this paper is organized as follows. In the first section, 'PWGLSynth signal model’, some problems in the current signal model are examined, focusing on audio analysis and music information retrieval. In the second section 'Optimizing signal processing’, a new, more general signal model is presented. The new model combines simplified patch programming with robust timing and efficient computation, allowing the user to combine different kinds of signals transparently. Finally, in the last section, 'Signals in a musical DSP system’, the proposed system is examined in the context of audio analysis.

2. THE PWGLSYNTH SIGNAL MODEL
PWGLSynth was designed for the scenario where Lisp-based user interface elements or precalculated sequencer events provide control data for a synthesis patch. We wanted to avoid splitting the system into audio rate and control rate paths, and developed the PWGLSynth refresh scheme which mixes output driven evaluation with data driven control events. In practice, boxes can get notified when their incoming control signal changes.

This notification is called a refresh event. A box can respond to a refresh event by performing some calculations that assist the inner audio loop. When audio outputs are connected to inputs that require refresh, the system generates refresh events at a global control rate. For example, an audio oscillator can be connected to a filter frequency control with the expected behaviour while still avoiding the need to recalculate filter coefficients at audio rate. When considering audio analysis, the scenario changes drastically. Control signals are generated from an audio signal, often in real time.

2.1. Prospects for audio analysis and music information retrieval
Audio analysis essentially involves a mixture of sparse event streams and fixed interval sample streams. Some analysis modules will recognize certain discrete features of an input stream, while others will retrieve some higher level parameter from an input stream, usually at a lower signal rate than that of the audio signal. A buffered FFT analysis might be triggered at a certain sample frame, resulting in a set of high level parameters that require further processing.
2.2. Towards a general unified signal model in a mixed rate system

The PWGLSynth Refresh scheme could in theory be adapted to suit audio analysis. A FFT box could store audio data internally until a buffer is full, then perform analysis and generate refresh events along with new output data. However, PWGLSynth provides no guarantees on the timing of refresh events generated during synthesis processing, as the scheme was devised for user interface interaction and automatic conversion of audio into control rate signal. The refresh scheme is well suited for simple control signal connections, but is not sufficient for the general case.

Since the refresh calls happen outside the synthesis processing, a unit delay may occur in the transition of the signal from audio to control rate. While not critical for user interface interaction, even a small scheduling uncertainty is not practical for audio analysis, where further processing will often be applied to the control signal. It is important to have well defined timing rules for the case of mixed signal rates.

Why not employ audio rate or the highest required rate for all signals? While this would guarantee robust timing, the very central motive for using a control rate at all is to optimize computation. Efficient handling of control signals can increase the complexity limit of a patch playable in real time, extend polyphony or reduce computation time for an offline patch.

3. OPTIMIZING SIGNAL PROCESSING

3.1. State-dependency analysis

The central theme in DSP optimization is always the same: how to avoid performing unnecessary calculations without degrading the output. The signal model and system scalability are a central design problem of any synthesis programming environment [7].

A simple example patch is given in Figure 1. In this patch, a set of sliders, labeled (1), represent the user interface. An intermediate power function (2) is computed based on the slider values, finally controlling the corner frequency of a filter (3).

When a human observer looks at the patch, it’s immediately obvious that some calculations are not necessary to perform at the audio rate. This finding is a result of a dependency analysis, aided by our knowledge of mathematical rules. Updating filter coefficients tends to be computationally expensive, not to speak of the power function. Yet, the coefficients ultimately depend on only the two values represented by sliders. In other words, the power function or the coefficients will not change unless the user moves one or both of the sliders. This can be expected to happen much more rarely than at the audio rate.

This is a very specific case that nevertheless represents the whole control scheme quite generally. The key idea is to note that certain signals do not change often and to avoid calculations whenever they don’t. Traditional modular synthesis systems have separated signals into audio rate and control rate signal paths. In this scheme, the programmer is required to explicitly state which signal rate to use for any given connection. Often, separate modules are provided for similar functions, one for each signal rate. A unified signal model on the other hand greatly decreases the time required to learn and use the system, as well as increases patch readability, often resulting in compact and elegant synthesis patches.

3.2. Functional computation scheme

In the previous example, the dependency analysis is easy to carry out because we know the power function very well. Its state only depends on the mantissa and exponent. It is less obvious what would happen if the box would be some other, more obscure PWGLSynth box.

The power function has an important property we might overlook since it is so obvious, yet carries deep theoretical meaning: it is strictly functional, meaning that it has no internal state. All power functions behave exactly the same, given identical input, at all times. This is unlike a recursive digital filter, which has an internal state that influences its output.

It turns out that many of the DSP operations can be carried out with functional operators. These include the common arithmetic and all stateless functions. When we extend the allowed operations by an unit delay, practically any known algorithm is attainable. By formulating the synthesis process as a functional tree expression, dependencies are easy to find. When a value in the patch changes, only the values downstream in the patch from it will need to be recomputed. Functional representation has many benefits, as shown in PWGL, Faust [4] or Fugue [2], all successful musical programming languages.

Traditional digital signal processing relies heavily on internal state, which can be represented by a filter buffer.
or a delay line feedback path. For the needs of dependency analysis, full functional rigor is not required. We only need to recognize that DSP modules with internal state will produce different output with identical input, but different moment in time.

By adding a ‘time’ source upstream of all modules with state we can make sure that by representing the time via a sample clock as a fundamental part of our functional computation tree, we can include modules with state. Correct, strictly functional behavior is ensured by functional dependency analysis once time is recognized as a parameter to all stateful modules.

3.3. State change propagation and synchronous updates

Our proposed system, aimed towards an intuitive yet efficient computation model, mixes aspects of input and output driven models. The patch is modeled with a number of endpoints which are, in effect, the state space of the system. Every point at which an user can tap into the patch is given a state. These states form a network, where each endpoint is connected to other endpoints by computational expressions. By utilizing functional building blocks, we can carry out dependency analysis and determine which states need to be recomputed when a certain state value is changed.

Thus the actual computation process resembles the output driven model, where computations are carried out when output is needed. The distinction is, however, that the output can ‘know’ when a recomputation is needed since it will be notified of updated input states. In effect, an output driven computation schedule is created for every input state of the system. A change in the input state then triggers the processing that updates its dependent states, in an input driven fashion.

A special case arises when several states must be updated synchronously. If each state update triggers recomputation of the relevant dependent states, and a state depends on several updated states, multiple updates are unnecessarily carried out. In addition, this would break any timing scheme that requires that updates happen regularly with the sample clock, such as a unit delay primitive.

This problem can be solved with update blocks that consist of two steps: first, all states that are about to be updated will be marked as pending. This pending status also propagates downstream in the state network, with each state keeping count of the number of pending states it depends on.

The second step consists of updating the state and releasing the pending flag. During release, all dependent states will decrement their pending counter and when it reaches zero, all states they depend on will have updated their value and the computation can be performed.

For efficiency, a third update mode is introduced. It will ignore the pending counters and just trigger computation and reset the pending counter to one. This mode is what will be used for the most frequently updated state, namely the sample clock. This allows for avoiding branches inside the innermost audio loop while marking all states that depend on the sample clock pending until the next input. Viewed within the entire model, the sample clock is always pending, apart from the moment that the actual audio rate computation is performed. This leaves the portions of the patch that don’t deal with audio available for control rate computations outside the inner audio loop.

3.4. Signal model overview

The complete synthesis scheme goes as follows: by default, boxes that depend on the sample clock are by default set to ‘pending’ with counter 1. Before updating the sample clock, all control event or user interface state changes are updated as a block and released. This triggers all calculations that do not depend on the sample clock. Finally, sample clock is updated. Since dependencies reflect on the whole patch downstream from a given box, the sample clock update is in fact the actual audio computation.

Intermediate states can be automatically inserted at patch points where signal rates mix. These states work as cached results of the lower rate signal process for the higher rate process.

4. SIGNALS IN A MUSICAL DSP SYSTEM

4.1. Coarse and fine time

To further ease box development, more than one clock source can be provided on the global level. Modules that require clock input could connect to either an audio rate or a control rate clock source. In most cases this connection should be made implicit and not visible to the user. A typical example with several different clock source possibilities would be an oscillator that functions as either an audio oscillator or a LFO for some computationally expensive operation.

It is possible to add metadata to the modules in order to automatically choose the most appropriate clock source for a given situation. If a sine oscillator is connected to an input that prefers a coarse control signal, it can automatically revert to a coarse clock and therefore produce an appropriate signal. Clocking could also be made an optional user parameter.

4.2. Signal rate decimation in audio analysis

Considering audio analysis from the perspective of the proposed paradigm, we encounter a further problem. Consider a typical buffered analysis scheme, where high level parameters are retrieved from a block of audio samples. This decimates the signal rate, as only one output value is produced per sample block.

Regardless, there is a functional relation between the audio data and the extracted parameters, implying that within the dependency rules outlined above, the high level analysis results must also be refreshed at audio rate. For modules that produce a control rate signal based on an audio rate signal, an intelligent scheduling scheme is required.
The system is completed by a sample-and-hold primitive, accessible to box developers. The primitive is able to break a dependency chain. It takes two inputs, updating its output to the first input when and only when the second input changes. This makes it possible for a box designer to instruct the scheduler to ignore an apparent dependency. This exception to the scheme causes some complications. When resolving the order in which states need to be refreshed when an upstream state changes, all dependencies must be traced through the sample-and-hold, even though the chain is broken for the first input. This is to enforce correct scheduling in the case where the decimated rate signal is merged back into the higher rate stream.

4.3. Redundant f0 estimation - a case study

The example patch in Figure 2 demonstrates the scheme in audio analysis.

Three f0 estimators with different frame size work in parallel for increased pitch detection robustness, driving a simple sine oscillator that follows the pitch of the audio input. The modules that depend on audio rate sample clock are sound-in and sine. The three FFT modules (1) therefore also depend on the sample clock, but break the downstream dependency since their output (the spectrum) changes with a lower rate.

The audio input to each FFT fills a ring buffer, without causing any output state refreshes. Timing is provided by assigning a coarse clock signals for each FFT module. The clock update is timed to provide an update when the ring buffer is full.

Upon a coarse clock update, the FFT computations are performed, f0 estimation is carried out and the median frequency is fed into the sine osc. However, since the sine osc depends on the fine clock, its operation is suspended and separately activated by the fine clock after the analysis step. Thus a delay-free and correctly scheduled audio-analysis - audio signal path is preserved with no waste of computational resources.

5. CONCLUSION

In this paper, we examined a signal graph processing system from a theoretical viewpoint. Some criteria for avoiding wasted computation operations were examined. We proposed a new signal model as a hybrid of two well known signal model schemes, which offers intuitive and robust system response with both sparse event streams and regular sample streams. The system features synchronic updates of input values as well as intelligent refreshing of output values. Finally, the proposed signal model was examined in the context of audio analysis.

6. ACKNOWLEDGEMENTS

This work has been supported by the Academy of Finland (SA 105557 and SA 114116).

7. REFERENCES


