The SysSon Platform

Technical Report TR-2016-09-1
Institute of Electronic Music and Acoustics, Graz
(Status: completed)

Hanns Holger Rutz

September 2016

1 Implemention of If-Branch

A small DSL is provided that allows the definition of conditional branches using a syntax relatively close to standard Scala 'if/else' blocks. The following shows the example scenario to test the implementation:

```
SynthGraph {
  val amp : GE = "amp" .kr(0.2)
  val freq: GE = "freq".kr

  val res0: GE =
    If (freq > 1000) Then {
      SinOsc.ar(freq)
    } ElseIf (freq > 100) Then {
      Dust.ar(freq)
    } Else {
      WhiteNoise.ar
    }

  Out.ar(0, Pan2.ar(res0 * amp))
}
```

We first have a **"monolithic"** implementation that rewrites this graph into one single UGen graph where all branches are always computed but the resulting signal `res0` only contains the signal of the "active" branch. The UGen graph is shown in Fig. 1. Basically the branch signals are multiplied by the logical branch condition (a graph element that is forced to be zero or one) and then summed up.

The second implementation then actually performs the **modular** decomposition into a set of related UGen graphs. Each conditional branch becomes a child UGen graph that will be run in its own `Synth` instance. Using a `Group` we can share the same control signals among them. For example, a control is specified for the "return bus" to which all branches are adding up their output. The main graph uses `Pause` UGens to start and stop the branches, receiving additional controls for the node-identifiers of the children. If a branch refers to elements from the outer context, auxiliary buses are established. In the example, the `freq` control signal is used by the main body (because it forms part of the branch-conditions that are tested here) and inside two of the three branches. Therefore, the main graph routes this signal to an auxiliary bus using an `Out` UGen, and that signal is then read by the respective child branches using an `In` UGen.

Several possibilities of handling the "return signal" have been evaluated, and the implementation settled on a simple and straight forward construction, whereby the property of the acyclicity of the directed graph implies that dependants on the branch signals can only occur after them in the sequence of graph elements registered with the `SynthGraph`. We thus initiate a new child branch after having visited any if-branch whose return type is `GE`. In other words, `Out.ar(0, Pan2.ar(res0 * amp))` will be encapsulated in another (forth) child branch.

1.1 IfLag

We want to be able to specify a fade-out time in seconds. We distinguish between fade-out phase and normal state. During fade-out phase, changes in conditionals have no effect. Once the fade-out is complete, the conditionals take effect again. This means that if the branch changes fast from one to two back to one, we might not actually pause the first branch and resume the second one, but simply fade out the first and then fade it back in (retrigger `ThisBranch`).

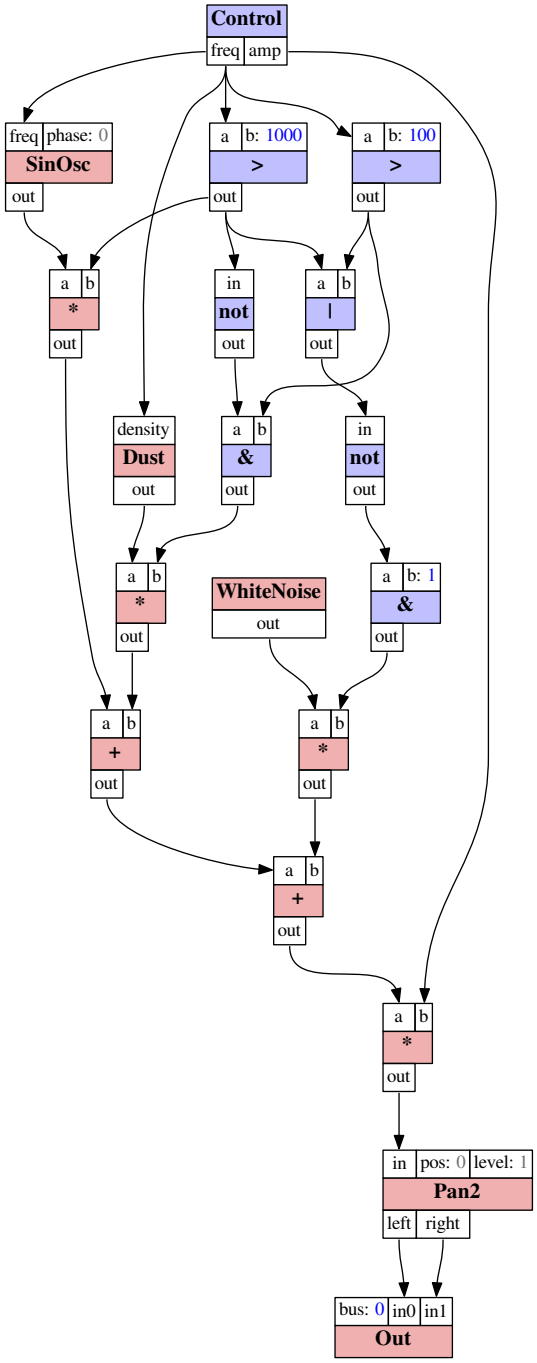
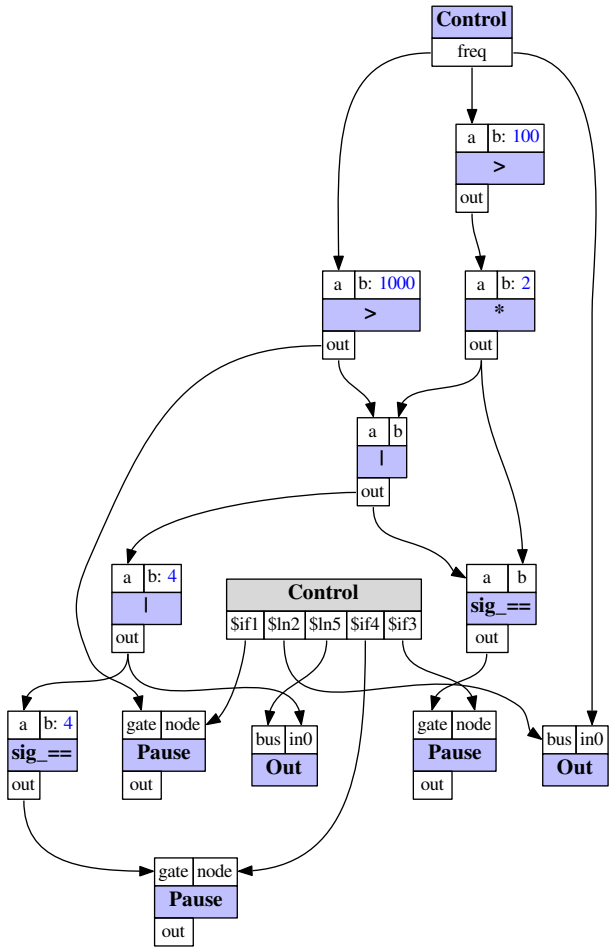
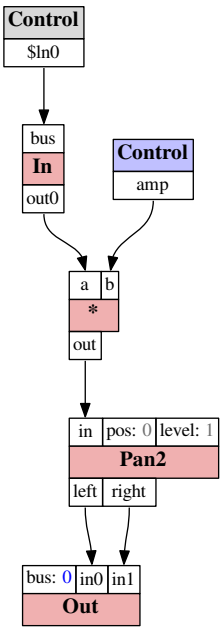


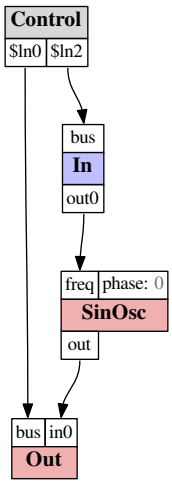
Figure 1: “Monolithically” expanded synth graph.



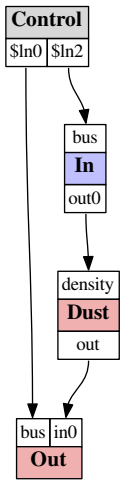
(a) Top level



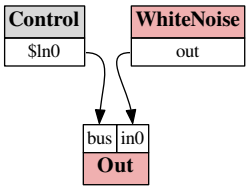
(b) Child 4 (top-level sink)



(c) Child 1 (if-branch)



(d) Child 2 (else-if-branch)



(e) Child 3 (else-branch)

Figure 2: Modular expanded synth graph.

The implementation changes from trigger to gate signal for `ThisBranch`. This is achieved by adding a release period of the given duration when the branch index changes. In this period, the old index is still held, but the gate signal goes low. Any changes in the index during the release are blocked out. Once the release has completed, the current index is let throw and causes a new branch to be resumed with a high gate signal.

Here is an example:

```
SynthGraph {
  val tr    = Impulse.kr(ControlRate.ir / 15)
  val ff    = ToggleFF.kr(tr)
  val dur   = ControlDur.ir * 5
  val res   = IfLag (ff, dur) Then {
    val gate  = ThisBranch()
    val env   = Env.asr(attack = dur, release = dur, curve = Curve.lin)
    val eg    = EnvGen.ar(env, gate)
    Seq(eg, DC.ar(0)): GE
  } Else {
    val gate  = ThisBranch()
    val env   = Env.asr(attack = dur, release = dur, curve = Curve.lin)
    val eg    = EnvGen.ar(env, gate)
    Seq(DC.ar(0), eg): GE
  }
  Out.ar(0, res)
}
```

The corresponding phase diagram is shown in Fig. 3. Because there is no release phase in the very beginning, the period where the first signal has a high gate is longer. Then one can see that the release of the previous branch does not overlap with the attach of the next branch. No two nodes are computed at the same time.

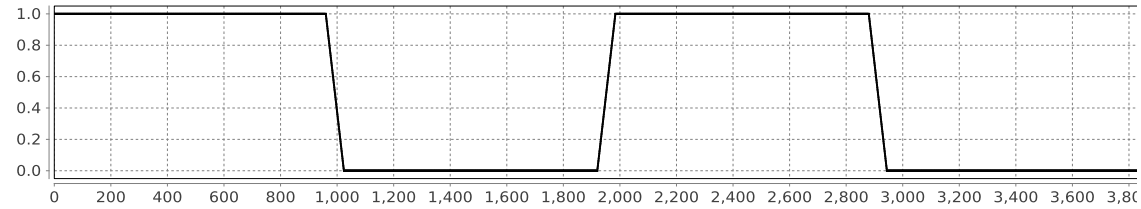
2 Integration of If-elements with Sound Processes

The UGen graph builder in *Sound Processes* was based on the basic building block in *ScalaCollider*, and operated in an "incremental" fashion: For each graph element expanded, it made a copy of the current state of the builder (controls, UGens expanded so far, etc.), then tried to expand the element. If in the course of this expansion a missing context property is detected, such as a missing attribute map entry, an exception is thrown, and the state is reset to the previous stable state.

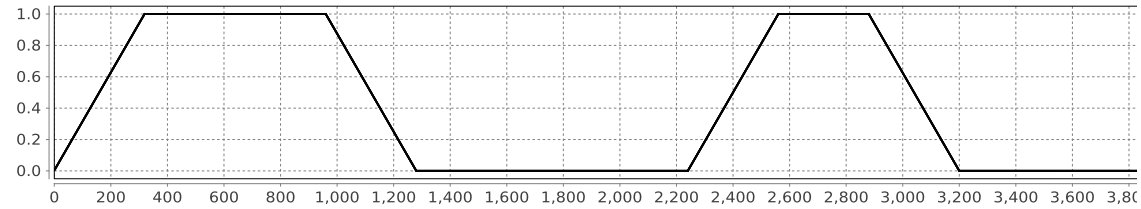
Incorporating the `NestedUGenGraphBuilder` from the *ScalaCollider-If* library makes this even more complicated, because now state goes as deep as updating the child branches of an if-block. While it is not impossible to preserve this state, the bookkeeping will become very large, and therefore we must assess the value of the incremental build.

The original concept of the incremental build was that two graphs could co-dependent on each other, i.e. there might be a situation where two graphs could only be completely expanded, if each had partial knowledge about the other, usually in terms of the number of channels used for some particular output or input. So the vision was that for example the first graph could be half built, producing some knowledge that would then be needed by the other graph, and the second half of the first graph could only be completed once the other graph successfully provided some information.

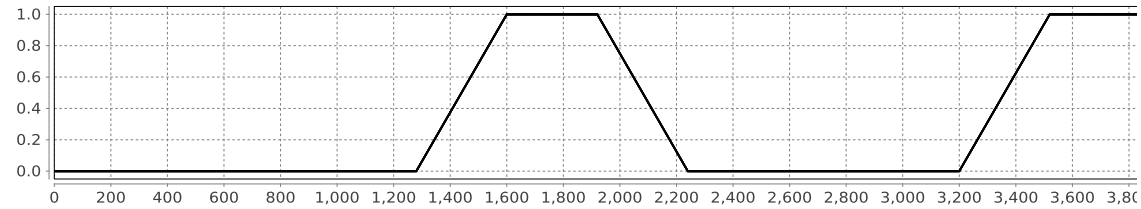
But is this really a common scenario? From observing debugging sessions, it can be said that:



(a) The flip-flop signal used as condition for the first branch
(the linear slopes are an artifact resulting from converting a k-rate to an a-rate signal)



(b) Gate signal of the If-Then branch



(c) Gate signal of the Else branch

Figure 3: Phase diagram for an IfLag block with two branches

- incremental build is prone to bugs, because updating correctly as new information enters the system is difficult, and rolling back is also difficult.
- often the element that requires the missing information is deep in the graph, which means that building the graph does not advance very far, so the advantage of having keeping already expanded elements becomes moot. It might be just as fast building the graph completely from scratch in the second and third attempt.

Therefore, if the incremental is dropped, we may get rid of a lot of complexity, and probably performance is not affected at all. The disadvantage is that large parts of the `AuralProcImpl` will have to be rewritten. On the other hand, we know that this class has bugs for many corner cases, and cleaning it up may benefit *SysSon* in the longer term.

Solution: We dropped indeed the incremental graph build in favour of a more simple approach where the arrival of missing attribute entries leads to an entire rebuild of the graph.

Artefacts have been published for the If-Then extension: `https://github.com/iem-projects/ScalaCollider-If`, and *Sound Processes* version 3.6.0 incorporates this into its own graph building.

2.1 Remaining Work

- We want to foresee the possibility to match for i-rate conditions that can be resolved at graph expansion time, for example because the condition evaluates to a constant number. In that case, we can avoid creating sub-graphs for each branch but simply synthesise the

branch that is always active. This exercise is not just a tweak for performance, but will be useful for evaluating constants coming from an `Obj`'s attribute map in *Sound Processes*.