A GRAMMAR FOR ANALYZING AND OPTIMIZING AUDIO GRAPHS

Vesa Norilo

Centre for Music & Technology
Sibelius Academy
Helsinki, Finland
vnorilo@siba.fi

ABSTRACT

This paper presents a formal grammar for discussing data flows and dependencies in audio processing graphs. A graph is a highly general representation of an algorithm, applicable to most DSP processes.

To demonstrate and exercise the grammar, three central problems in audio graph processing are examined. The grammar is used to exhaustively analyze the problem of scheduling processing nodes of the graph, examine automatic parallelization as well as signal rate inferral.

The grammar is presented in terms of mathematical set theory, independent of and thus applicable to any conceivable software platform.

1. INTRODUCTION

Most signal processing algorithms are extremely well suited to be represented by graphs, connected networks of nodes. The nodes in the network correspond to processing operations, while the interconnections denote signal flow.

In addition, audio graphs are typically directional. Signal flows traverse the graph from initial sources to eventual destinations, entering processing nodes via inputs and exiting them from their outputs.

Considering how the graph metaphor is so general and widely applicable, it would be highly beneficial if a formal language could be used to reason about graphs in the context of analyzing and transforming audio algorithms.

This paper employs the elementary principles of mathematical set theory in discussing, analyzing and transforming audio graphs. Some straightforward additional notation is introduced to simplify the discussion about data dependencies and node reachability.

As performance is typically critical in audio applications, an immediate field of interest is using the emerging theoretical identities to optimize the computation of audio graphs. The emerging logical language is employed to present and discuss proofs about scheduling and transforming audio graphs, without tying the results to a particular platform or system. The research has been applied to the foundations of the author's work with signal processing compilers[1].

The rest of this paper is organized as follows. First, in Section 2, *Notation*, elementary operators for describing subgraphs and supergraphs are introduced. In Section 3, *Scheduling a DAG*, execution schedule constraints for an audio graph are formally laid out. Section 4 *Parallelization*, discusses rules for automatic parallelization of an audio graph. Section 5, *Signal Rate Optimization* examines how graphs can be analyzed for required update rates. Finally, Section 6, *Conclusions*, summarizes the paper and the grammar introduced in it.

2. NOTATION

The study of graphs is a relatively recent but growing topic in mathematics. The type of graph best suited for digital computation of audio signals is the directed acyclic graph or DAG[2].

DAGs incorporate the direction of signal flow, so that outputs are fed into inputs, and prohibit cycles in the graph – necessary for a graph to be finitely computable.

Let us begin by defining set-theoretic operators and concepts to enable the analysis of DAGs. For an overview of elementary set theory, the reader is referred to literature[3].

2.1. Reachability

Reachability between two nodes is an intuitive concept. If there exists a path between the nodes, from node to node via connections, the nodes reachable from each other. Let this condition be represented by the general reachability operator, $a \updownarrow b$, that produces a boolean truth value.

2.1.1. Upstream Locator

Reachability becomes more useful if the path is constrained. Let us define a more specific reachability operator, the *upstream locator*. Stated as $a \uparrow b$, the operator is true if *a* can be reached from *b* by traversing the graph *upstream* – the direction opposite to the signal flow.

2.1.2. Downstream Locator

The inverse of the *upstream locator* is the downstream locator. Used for convenience, the operator can be defined simply as

$$a \downarrow b = b \uparrow a \tag{1}$$

.

2.2. Subgraphs and Supergraphs

In general graph theory, a subgraph is a set of nodes and interconnections that can be obtained from a larger graph by severing some of the connections. In this paper, we shall adopt a narrower definition of a subgraph.

Let us define a subgraph in terms of data dependency; let a subgraph consist of a particular *root node*, and any nodes reachable from it by traversing the DAG upstream. In other words, the

subgraph is the portion of the DAG through which signal must pass before reaching the input of its root node.

Let a supergraph be the opposite of subgraph. Let the supergraph consist of a root node and all the nodes that can be reached from it by traversing the DAG downstream.

Let a be an arbitrary node in a DAG. Using the locator operators, the subgraph of a can be defined as a section of the universal set $\mathbb{U}$

$$\Uparrow a = a \cup \{x \in \mathbb{U} : x \uparrow a\} \tag{2}$$

Likewise, the supergraph of a is

$$\Downarrow a = a \cup \{x \in \mathbb{U} : x \downarrow a\} \tag{3}$$

Supergraphs and subgraphs have an inverse relation;

$$b \in \Uparrow a \iff a \in \Downarrow b \tag{4}$$

Nested subgraphs and supergraphs imply subsets and supersets;

$$b \in \Uparrow a \Leftrightarrow \Uparrow a \supseteq \Uparrow b \tag{5}$$

$$b \in \Downarrow a \Leftrightarrow \Uparrow a \subseteq \Uparrow b \tag{6}$$

2.3. Summary of Notation

Notation	*Meaning*
$a \uparrow b$	*Node a can be reached from b by traversing the graph upstream*
$a \downarrow b$	*Node a can be reached from b by traversing the graph downstream*
$\Uparrow a$	*The set of a and all nodes x for which $x \uparrow a$ holds*
$\Downarrow a$	*The set of a and all nodes x for which $x \downarrow a$ holds*

3. SCHEDULING A DAG

Let $\mathbb{G}$ be a DAG describing an audio algorithm, consisting of several independent processing nodes. Should this DAG be transformed into a computer program, the first requirement would be to produce a correct processing order for the nodes.

The fundamental scheduling constraint is that the processing of a node can not commence before all its inputs are ready for processing. Let us define two informal operators, *Ready*, indicating whether a node can be processed or not, and *Finished*, indicating whether the node has been processed already. This results in;

$$Ready(a) = \neg(\exists x \in (\Uparrow a \setminus \{a\}), \neg Finished(x)) \tag{7}$$

Note that nodes with no input connections, ie. $\Uparrow a = \{a\}$ are always ready as they have no dependencies.

A linear list could be constructed from the nodes in graph $\mathbb{G}$ by sorting them according to reachability. A sorting algorithm that operates with a binary less-than predicate could be employed. By utilizing the upstream locator operator as the less-than comparison, a correct schedule can be constructed;

$$a < b \iff a \uparrow b \tag{8}$$

Such sorting algorithms are available in most programming languages, including the standard template library for C++[4]. Simply iterating through such a sorted list is guaranteed to process all the nodes in correct order, provided the sorting algorithm is compatible. This point is expanded in the following subsection.

3.1. Ordering and Reachability

It could be tempting to extend the semantic equivalence of the upstream locator to the full trichotomy of comparison operators;

$$a < b \iff a \uparrow b \tag{9}$$

$$a > b \iff a \downarrow b \tag{10}$$

$$a = b \iff \neg(a \uparrow b \vee a \downarrow b) \tag{11}$$

However, the metaphor breaks down at equality. Consider:

$$\begin{aligned} a &\in \Uparrow b \\ c &\notin \Uparrow b \\ c &\notin \Downarrow b \end{aligned} \tag{12}$$

This would give $a = c, b = c$ but also $a \neq b$, thus the hypothetical equality operator doesn't function as expected.

For less-than predicate sorting to work, the sorting algorithm must not rely on equality derived as in equation 11. The class of acceptable algorithms perform *strict-weak ordering*[5], which relies on a binary less-than operator.

As there may be more than one correct order for any strict-weakly ordered set, the exact result will depend on the sorting algorithm.

4. PARALLELIZATION

As stated in Section 3, there are typically several valid processing schedules for a DAG. In such cases, the ambiguity results from the fact that there are operations that are independent of each other. These operations can be performed in any order or even concurrently.

Any nodes that can be parallelized must therefore be ambiguously ordered. In other words, the nodes should satisfy strict-weak ordering according to upstream reachability in either order. Otherwise, one of the nodes is upstream reachable from the other, and the nodes must be serially processed to honor all the dependencies.

The condition for parallelization can thus be formally stated;

$$\begin{aligned} Parallelizable(a, b) &\iff \neg(a \uparrow b \vee a \downarrow b) \\ &\iff \neg(a \uparrow b \vee b \uparrow a) \\ &\iff \neg(b \in \Uparrow a \vee a \in \Uparrow b) \end{aligned} \tag{13}$$

More generally, two entire subgraphs are parallelizable if their intersection is the null set;

$$\Uparrow a \cap \Uparrow b = \emptyset \tag{14}$$

If this condition is met, the subgraphs can be processed independently from each other. Note that this only applies to the narrow definition of a subgraph as defined in Section 2.2, not the subgraphs of general graph theory.

If the intersection yields a non-empty set, the parallelizable components are the relative complements of the subgraphs and their intersection;

$$C = \Uparrow a \cap \Uparrow b \tag{15}$$

$$A' = \Uparrow a \setminus C \tag{16}$$

$$B' = \Uparrow b \setminus C \tag{17}$$

$$\tag{18}$$

Therefore, A' and B' can be executed in parallel, but C must be executed before either of them.

4.1. An Algorithm for Parallelization

As there is significant scheduling overhead in parallel computation, it is typically ideal to parallelize as little as possible while still maintaining full utilization of computing resources. A well known method for balancing utilization and overhead is the data flow work queue, where a central pool of available tasks is maintained. Each worker thread pulls a task from the repository, completes it and places any newly available tasks into the pool. Tasks become available as all their inputs are finished, as shown in equation 7.

Inadequate load balancing may cause performance degradation in the case of the data flow work queue. This means that some computational cores are performing useful work while some are not, possibly waiting for tasks that depend on the ones currently being processed. In the case of a general audio DAG, load balancing can be improved by increasing work item granularity – in other words, including fewer processing nodes in each work item. This in turn can cause the scheduling overhead from the growing number of work items to eradicate any gains made from improved load balancing.

Utilizing the results shown above, an algorithm can be constructed that automatically generates a parallelized work schedule. A work item size parameter is introduced to allow for fine tuning the tradeoff between load balancing and scheduling overhead. This algorithm carries the assumption that the computational time consumed by processing a set of nodes can be approximated or measured. The *size* of the work item corresponding to that set is proportional to the computational time.

- Start parallelization from a root node R.
- Obtain the subgraphs of all nodes connected to the inputs of R. Let these be $\{\mathbb{P}_1, \mathbb{P}_2, ...\mathbb{P}_n\}$. Let $\mathbb{I}$ be the index set $\{x \in \mathbb{Z} : 0 < x \leq n\}$.
- The serial dependency is

$$\mathbb{S} = \bigcup_{i \in \mathbb{I}} \left(\mathbb{P}_i \cap \bigcup_{j \in \mathbb{I}, j \neq i} \mathbb{P}_j \right) \tag{19}$$

- The parallelizable portions of the DAG are $\{\mathbb{P}_1 \setminus \mathbb{S}, \mathbb{P}_2 \setminus \mathbb{S}, ..., \mathbb{P}_n \setminus \mathbb{S}\}$
- The parallelizable portions exceeding the size of the chosen work item size treshold are kept. Those that fall below the treshold should be combined, largest with the smallest, until there are no more work items below the treshold or just one item is left.
- For any set $\mathbb{P}_k$ for which $\mathbb{P}_k \cap \mathbb{S} \neq \emptyset$, the respective parallelizable portion $\mathbb{P}_k \setminus \mathbb{S}$ has a dependency on $\mathbb{S}$, and must be scheduled only after $\mathbb{S}$ is entirely completed.
- If the serial dependency $\mathbb{S}$ exceeds the work item size, it should be recursively parallelized. Let the set of nodes $\{x \in \mathbb{S} : (\Downarrow x) \cap \mathbb{S} = \emptyset\}$ form the root set from which a new set of $\mathbb{P}$ subgraphs be built. $\mathbb{S}$ should be considered completed only when the newly parallelized tasks are all completed.

5. SIGNAL RATE OPTIMIZATION

Whereas parallelization is more concerned about when a node set *can* be processed while maintaining data flow integrity, signal rate optimization is about deducing when it *must* be processed.

Not all signals need equally frequent updates, and often significant efficiency can be gained by updating certain node sets at a lower rate. This optimization technique has a strong tradition, with the concept of *control rate* being central in many music software environments ever since the venerable CSound[6].

Another – arguably more desirable – approach is to analyze the signal paths in the audio DAG and automatically determine the desired signal rates for the most typical scenario. This approach, presented here in the terms of the set-theoretic approach of this paper, has previously been described by the author[7].

The automatic process can be guided by inserting non-processing nodes into the DAG whose sole purpose is to guide the signal rate inferral. Once the inferral is completed, these nodes can be removed from the DAG to avoid any overhead.

There are two kinds of sources, streaming and event-based. A streaming source will emit a sampled signal with regular sample intervals. Event-based sources react to some external or internally derived stimulus, producing an update upon receiving, for example, a MIDI event.

In both cases, it is desirable to process the supergraph of a source node according to its update rate. A filter processing the output of an oscillator should work at the same signal rate. Likewise, if an event-driven signal like an user interface slider seldom changes, computations that depend on it should be avoided when unnecessary.

To infer the DAG signal rates, signal sources must be identified. In an audio DAG, these sources are oscillators, audio file players, external audio inputs, user interface control signals and other inputs such as MIDI or OSC[8].

5.1. Source Discovery

A first step in the analysis of the required signal rate for a particular node is to discover which source nodes have the node in their supergraphs. According to equation 3, this can be determined by collecting the source nodes from the subgraph of the node. Let $\mathbb{SRC}$ be the set of all source nodes. As a starting point, we could assume that the node needs to be recomputed whenever one of its sources gets updated.

$$Sources(a) = \Uparrow a \cap \mathbb{SRC} \tag{20}$$

5.2. Source Arbitration

There is, however, a further consideration. Stateful processes such as filters or delay lines should be updated only according to their audio signal inputs. This ensures a steady sample rate which would otherwise be compromised by additional updates forced by control signals. Therefore, if a filter node has both an audio input and a

user interface slider in its supergraph, it should ignore the updates by the slider and only update according to the audio rate.

This can be solved by introducing *source priorities*. By providing strict weak ordering[5] of the sources, the desired behavior can be attained. If the audio input has a higher priority than the user interface element, nodes that have both sources should just ignore the user interface updates until the next audio-driven update happens.

5.3. Update Regions

After arbitration, the nodes should be classified according to the arbitrated sources driving them. In fact, both arbitration and classification can be described by simple equations. Let S_{audio} be a high priority source, followed by a medium priority S_{OSC} and a low priority S_{UI}. These correspond to audio, OSC-event and user interface update priorities. Let the node sets they drive be $\mathbb{G}_{audio}$, $\mathbb{G}_{OSC}$ and $\mathbb{G}_{UI}$. This gives;

$$\mathbb{G}_{audio} = \Downarrow S_{audio} \tag{21}$$

$$\mathbb{G}_{OSC} = \Downarrow S_{OSC} \setminus \mathbb{G}_{audio} \tag{22}$$

$$\mathbb{G}_{UI} = \Downarrow S_{UI} \setminus (\mathbb{G}_{audio} \cup \mathbb{G}_{OSC}) \tag{23}$$

This list could further be expanded on the simple principle that each source drives all the nodes in its supergraph, except those that also belong to a supergraph of a higher priority source.

Such a system of graphs can be scheduled easily by processing the node sets in reverse order of priority. From equations 4 and 23 it can be deduced that;

$$\mathbb{G}_{audio} \cup \mathbb{G}_{OSC} = \Downarrow S_{audio} \cup \Downarrow S_{OSC} \tag{24}$$

$$\mathbb{G}_{UI} \cap \Downarrow S_{audio} = \emptyset \tag{25}$$

$$\mathbb{G}_{UI} \cap \Downarrow S_{OSC} = \emptyset \tag{26}$$

$$\forall x \in \mathbb{G}_{UI}, (\Uparrow x) \cap \mathbb{G}_{OSC} = \emptyset \tag{27}$$

$$\forall x \in \mathbb{G}_{UI}, (\Uparrow x) \cap \mathbb{G}_{audio} = \emptyset \tag{28}$$

Thus, no subgraph of any node in $\mathbb{G}_{UI}$ can contain any nodes that also belong to $\mathbb{G}_{OSC}$ or $\mathbb{G}_{audio}$. Therefore, $\mathbb{G}_{UI}$ can safely be scheduled before the higher priority blocks. The proof can be extended to show that any lower priority node group can always be safely scheduled before higher priority node groups. This is especially important in the case where both sources are driven from a coherent clock source, such as traditional audio and control signals. Failure to schedule coherent clock sources according to their priority would result in potential undesired delays at signal rate boundaries.

5.4. Priority Inversal

In many algorithms, more precise control of signal rates is required. With just the inferral system described so far would make it impossible, for example, to derive MIDI events from audio signals at any rate below the audio sampling rate. To solve this problem, it is necessary to be able to override the source priorities locally, at a specific DAG junction.

The best possible priority inversal mechanism is still a topic of active research. As an initial solution, priority escalation is suggested. A special source could be generated for any DAG junctions where priority inversal is desired. This source would have a higher priority than the one it is meant to override. Scheduling-wise, this additional source should be processed according to the reverse priority order, generating some additional bookkeeping overhead.

6. CONCLUSIONS

In this paper, elementary concepts for formally discussing directed acyclic graphs in audio context were introduced. These concepts include the upstream and downstream reachability operators, as well as the construction of subgraphs and supergraphs. Taken together, these devices facilitate set-theoretic discussion of processing directed acyclic graphs for audio signals.

Three practical, highly important problems were examined using the newfound grammar. The problem of scheduling operations described as a signal processing graphs was examined and deemed a strict-weak order based on a simple reachability operator. The ambiguity of that strict-weak order was leveraged to analyze the problem of concurrently executing portions of an audio processing graph. Finally, the grammar was utilized to discuss automatic signal rate optimization and discover the additional scheduling constraints such a system imposes.

The concepts form the basis of the author's work on signal processing languages and compilers[1]. They are presented here independently from any programming language or system, instead employing the notation of mathematical set theory. The concepts are not overwhelmingly difficult, but utilization of formal grammar helps avoid ambiguously worded statements and pseudo-rules. Further, statements in a formal language lend themselves to further reasoning, identities and proofs. This is of vital importance when constructing compilers and interpreters, especially in the case of automatic parallelization of user algorithms. This paper is written in the hopes of providing assistance in the form of a grammar to the researchers working with these problems.

7. REFERENCES

[1] Vesa 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, NUIM.

[2] F Harary, Robert Z Norman, and D Cartwright, *Structural models: An introduction to the theory of directed graphs*, Wiley, 1965.

[3] H B Enderton, "The Joy of Sets. Fundamentals of Contemporary Set Theory.," *The Journal of Symbolic Logic*, vol. 59, no. 4, pp. 1441, 1994.

[4] Alexander Stepanov and Meng Lee, *The Standard Template Library*, Number X3J16/94-0095, WG21/N0482. Prentice-Hall, 1995.

[5] Bernd Schröder, *Ordered Sets: An Introduction*, Birkhäuser Boston, 2002.

[6] Richard Boulanger, *The Csound Book*, vol. 309, MIT Press, 2000.

[7] Vesa Norilo and Mikael Laurson, "Unified Model for Audio and Control Signals," in *Proceedings of ICMC*, Belfast, Northern Ireland, 2008.

[8] Matthew Wright, Adrian Freed, and Ali Momeni, "OpenSound Control: State of the Art 2003," *Time*, pp. 153–159, 2003.