# Synchronous Data Flow Modeling for DMIs

Danielle Bragg
Department of Computer Science
Princeton University
dbragg@princeton.edu

## ABSTRACT

This paper proposes a graph-theoretic model that supports the design and analysis of data flow within digital musical instruments (DMIs). The state of the art in DMI design does not provide standards for the scheduling of computations within a DMI's data flow. Without a theoretical framework, analysis of different scheduling protocols and their impact on the DMI's performance is extremely difficult. As a result, the mapping between the DMI's sensory inputs and sonic outputs is classically treated as a black box. DMI builders are forced to design and schedule the flow of data through this black box on their own. Improper design of the data flow can produce undesirable results, ranging from overflowing buffers that cause system crashes to misaligned sensory data that result in strange or disordered sonic events. In this paper, we attempt to remedy this problem by providing a framework for the design and analysis of the DMI data flow closely modeled after a framework for digital signal processing. We also propose the use of a scheduling algorithm built upon that framework, and prove that it guarantees desirable properties for the resulting DMI.

## Keywords

DMI design, data flow, mapping function

## 1. INTRODUCTION

Designing the data flow of a DMI is a complex task. A DMI typically takes input data from sensors and passes this data through a set of functions. This set of functions, the paths along which data is directed from one function to the next, data buffering along these paths, and the timing of the function executions constitute a data flow design.

To build a system, the designer must determine how data is directed through the system; where and how to store data temporarily in the data flow; how to encode delays; how to fuse data from diverse sensor sources; and how varying computational times for the functions involved impact system performance. Mistakes in the data flow design have varying impact on DMI performance. Some mistakes can result in nuanced changes, while others result in obviously undesirable performance, or even system crashes. Designing robust data flows will only become more difficult as system complexity increases, and yet no framework is currently employed in state-of-the-art DMI design to help designers navigate the design of these complex systems.

Our work provides this crucial infrastructure, as well as a scheduling algorithm for the data flow with desirable performance guarantees. It forms the foundation for new instrument-building tools that are capable of appropriately automating choices about when to trigger computations. Further, this approach to automating choices about computation triggering is demonstrably suitable for a very wide set of potential DMI designs. For example, it can handle DMIs with varied data sources, DMIs with deep data flow topologies, and DMIs with rampant latency and buffering considerations. A design tool supported by such a scheduling algorithm would both enable more efficient, less error-prone experimentation with data flow designs and facilitate building more complicated instruments that are not currently practical for designers working with ad hoc approaches.

### 1.1 Meeting Performance Requirements

Data flow design is complicated by a number of technical and artistic requirements. Since a DMI is used for artistic purposes, there are no universal artistic requirements. Nonetheless, there are some natural performance requirements that a designer or musician might have for a DMI. Our framework for DMI data flow design facilitates design analysis to ensure that such criteria are met.

A basic performance requirement is that the system does not crash. System crashes can be caused by faulty buffer management in the data flow. As data flows through the DMI's pipeline, small queues typically store the data until it can be used. Bounded queue size is required to prevent overloading the system's memory and causing the system to crash. Even if a system crash does not occur within the time period that the instrument is in use, buffer growth increases delay since data must remain in the system longer before it is output. When data that passes through unbounded buffers meets data that passes through bounded buffers, increasing misalignment is also likely to occur.

Another minimal performance requirement is that the DMI produces sound according to the basic functionality intended. For example, consider a piano-inspired DMI controlled by hand motions. Hand position, detected by processing Kinect data, maps to pitch. Acceleration, detected by an accelerometer attached to the hand, maps to volume. Deducing hand position from the Kinect data takes more time than minimally processing the raw accelerometer data. As a result, a naive data flow design that ignores these computational differences will mismatch data to produce pitches at the wrong volumes (or volumes at the wrong pitches). Similarly, varied data collection rates across sensor inputs can contribute to undesirable sonic outputs. Synchrony is not always required, but is certainly the desired behavior in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*NIME'13,* May 27 – 30, 2013, KAIST, Daejeon, Korea.
Copyright remains with the author(s).

some contexts and is difficult to enforce. The algorithm we suggest in this work makes synchrony enforceable.

Latency and jitter should typically be minimized or bounded in order for the tool to serve as an effective expressive tool. Latency is defined as the processing delay between a physical control and the resulting sonic event. Jitter refers to variability in latency due to slight variations in data processing and transmission times. A very general guideline is that latency should be under 10 ms, with jitter ±1 ms [21]. This extremely low bound on latency and jitter requires efficient DMI data flow design and scheduling.

It is often desirable for DMIs to control synthesis events sequentially. To define sequential control, suppose there are two control inputs $c_1$ and $c_2$ generated at times $t_1$ and $t_2$ respectively, where $t_1 \leq t_2$. Also suppose that $c_1$ triggers sonic response $r_1$, and $c_2$ triggers response $r_2$. If $r_1$ happens at time $t_3$ and $r_2$ happens at time $t_4$, we seek $t_3 \leq t_4$. Sequential control is guaranteed by all acoustic instruments since the performer's movements directly generate the responding sound. Realistic digitization of acoustic instruments must preserve this property. In general, if sequential control is not guaranteed, the instrument runs the risk of not responding according to the user's expectations.

It is difficult for DMI designers to guarantee these performance requirements because there are no standard scheduling protocols or theoretical framework for analyzing the data flow through a DMI. In this absence of standards, DMI builders often implement naive scheduling protocols. For example, the built-in protocol in Max/MSP executes a computational node whenever a particular input is received. Such scheduling protocols come with no guarantees about system performance. In order to evaluate and refine our protocols, we need a theoretic framework within which to work. In this paper, we propose just such a framework, as well as a scheduling algorithm with performance guarantees.

## 1.2 Contribution

As we have discussed, the design of a DMI data flow is often complex. The designer must meet both technical and artistic performance criteria through his design and scheduling of the DMI data flow. In order to guarantee performance, DMI builders must be able to analyze the data flow explicitly. Consequently, a framework for designing, scheduling, and analyzing the data flow becomes crucial.

In this paper, we propose such a framework in a graph-theoretic model of the DMI data flow. We propose extending Lee and Messerschmitt's graph-theoretic framework for modeling data flows in digital signal processing [12] to model DMI data flows, and demonstrate its power in doing so. When applied to DMIs, this model supports efficient and accurate design and analysis by facilitating the following:

- Identification of relationships between inputs and outputs.
- Analysis of system performance under various scheduling algorithms.
- Development and deployment of scheduling algorithms that guarantee desired system performance.

We also propose applying Lee and Messerschmitt's PASS scheduling algorithms to DMIs, and prove that these algorithms produce DMIs with the following properties:

- The DMI can run forever with bounded buffer size.
- Whenever a computational component is scheduled to execute, sufficient input data is available.
- The DMI performer has sequential control over the instrument outputs.
- Heterogeneous sensor collection rates are handled safely.

## 1.3 Related Work

DMI builders and researchers have focused on some portions of the DMI data flow design separately. In particular, the mapping function that embodies the relationship between controls and synthesis events has been explored [19, 7, 8]. In that work, researchers present high-level organizations and properties of the mapping space, but not the detailed picture required for implementation-level design and analysis. Sensor fusion is another challenge that many DMI builders must face. When diverse sensors gather data about the same physical event, it is often difficult to synchronize data gathered by different devices. System-specific solutions have been developed with varying degrees of success [16, 14, 20]. However, because these solutions are non-generalizable, DMI designers who wish to create systems that can be controlled by any type of controller must solve this problem from scratch. Systems for collaborative development and networked playing of DMIs [15, 14] highlight these design problems. As these systems move from local area networks to the internet, the larger distances increase latency and jitter and amplify the effects of scheduling protocols [3]. A framework that supports the design of the entire data flow would allow for the development of generalized solutions for all parts of the DMI data flow design.

To aid in the implementation of data flow for DMIs, work has been done on the design of the data that passes through interactive systems. Open Sound Control (OSC) protocol was developed with DMI construction in mind [18]. While OSC provides infrastructure for piping data through a system, it does not provide guidelines for the scheduling of that data. Programming languages and environments like Max/MSP still leave scheduling problems largely unsolved and up to the programmer [17]. A Max/MSP programmer can create complex patches, but he is still responsible for ensuring proper data flow within the patch. Max programming requires careful planning to ensure that an object's computation is triggered at the left inlet only once other inlets have received appropriate data, and that the flow of data from an object's outputs to the "downstream" objects in the patch continues to trigger computations in a sensible order. Even existing toolkits like MnM [4] do not fully address this problem of data flow design. The community needs standards and models to facilitate more complex projects without overburdening the designer. In this work, we attempt to fill this need for DMI data flow design.

Problems of data flow organization and analysis are not unique to DMIs. Indeed, any field where data streams through a system in a non-trivial course must face similar problems. For example, data flow modeling and analysis also has strong applications in low-level systems operations (e.g., [5]). More closely related to DMIs, data flow has also been richly explored within the field of digital signal processing (DSP) [13, 12, 10]. In that work, various graphs are used to model DSP operations. Nodes represent computational units, while directed edges represent data passing from one computation to the next. Much of that work focuses on the scheduling of the computational units so that certain conditions are met. [2] proposed a scheduling protocol specialized for time-triggered multimedia systems, but not for human-controlled DMIs. In our work, we exploit the close relationship between DSP and DMIs to extend synchronous data flow modeling, a canonical framework designed for DSP algorithms, to DMI data flows. Examples of DMIs that fit the synchronous data flow model include an instrument that uses biosignals to constantly produce sound and an instrument that continuously produces sounds based on a dancer's position on the stage. Using this framework, we analyze DMI designs and provide new theorems

about DMI system performance under a particular class of scheduling algorithms.

## 2. THE MODEL

In this section, we propose our graph-theoretic model of the data flow of a DMI. The model is heavily based on the foundational work of Lee and Messerschmitt [12], whose data flow model is designed especially for DSP. Since DSP operations are often part of a DMI's data flow, the model naturally extends to the DMI's data flow. We begin by describing basic features of the data flow model presented by Lee and Messerschmitt that we incorporate in our work.

We represent the DMI as a directed graph $G = (V, E)$, where $V$ is a set of nodes (called blocks in [12]), and $E$ is a set of directed edges (called arcs in [12]). Each node represents a computational function invoked by the DMI. Edge $(n_i, n_j)$ from node $n_i$ to node $n_j$ represents the fact that $n_i$'s computations produce data that is directly consumed by node $n_j$. Figure 1 shows the graph representing a data flow involving three independent operations. The first operation receives inputs, and pipes outputs to two subsequent operations. Each of those operations also produces some output. A sample input or output unit is a numerical value or a vector of values.

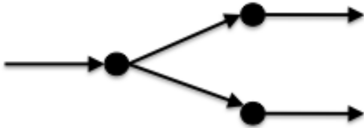

**Figure 1: Simple data flow graph.**

A node is called "synchronous" if we know a-priori how many units of inputs it will consume and produce for each computation that it performs. On a synchronous graph, each edge leading into a given node is labeled with a number. This number represents the number of data samples the computation must take in along each channel to perform a computation. We use the word "channel" to refer to a contiguous sequence of edges, or directed path, in the graph. Each edge leading out of a given node is also labeled with a number. Note that these numbers will always be whole numbers, since the concept of a fraction of a data sample does not make sense. Conversely, an "asynchronous" node is one where the number of inputs and outputs is data-dependent. In a DMI, a synchronous node might be responsible for scaling data collected by a particular sensor. An asynchronous node might be responsible for filtering that data, only passing on the incoming data if it satisfies a particular condition. For simplicity, we focus on synchronous data flows in this work. Figure 2 depicts a simple data flow graph, with specified computation input and output requirements.

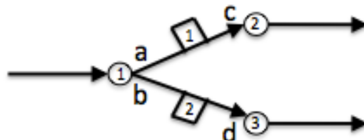


**Figure 2: Simple synchronous data flow graph with displayed computational inputs and outputs.**

Any synchronous graph can be represented as a matrix, called a "topology matrix" $M$, where $M(i, j)$ represents the data flow along edge $i$ w.r.t. node $j$. If data flows into node $j$ along edge $i$, this is a positive number of inputs. If data flows out of node $j$ along edge $i$, then this is a negative number. And if edge $i$ is not directly connected to node $j$, then this is 0. The topology matrix for the synchronous graph given in Figure 2 is matrix

$$\begin{pmatrix} a & -c & 0 \\ b & 0 & -d \end{pmatrix}.$$

Each edge can be replaced by a queue used to store data passed between nodes. Buffer sizes will inevitably vary over time. We will use $b(t)$ to represent the vector of queue lengths at time $t$. Lee and Messerschmitt model each edge as a first-in-first-out (FIFO) queue, and we do the same. Alternative queueing strategies exist, and we will discuss their performance consequences in Section 5.

## 3. MODELING A DMI

At a high level, a DMI operates by streaming data through a sequence of computational components. Each component performs a specified set of computations on the inputs to produce a set of outputs that are piped to the next component. Because the data flow is primarily linear through this sequence, the basic graph structure we use to represent a DMI is a multi-layered graph. Each layer in the graph corresponds to a sequential component of the linear data flow. We can further decompose each component into a set of independent functions. Each independent function is represented as a node in the graph at an equal depth in the graph. If a given component is not decomposable in this way, then it is simply represented by a single node. Figure 3 depicts this basic layered graph structure for a DMI.

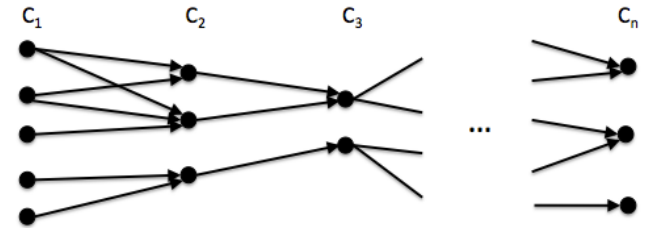


**Figure 3: Data flow model of an arbitrary DMI with $n$ sequential computational components.**

Though a layered model of a DMI's data flow is practical and useful for many purposes, it is not entirely correct. In many DMIs, the computational components are not strictly ordered [18]. Specifically, feedback loops might stream data produced by a particular component to a preceding component. Conversely, some data might bypass one or more components, thereby skipping layers in the layered model.

We can easily expand our layered graph to model DMIs with non-linear pipelines. Figure 4 depicts an example of a DMI data flow where the layers are not strictly ordered. Specifically, it contains one edge that acts as a feedback loop, and one edge that bypasses layers of the data flow, to pipe data directly to the final layer.

The model can also represent delays within the data flow and inputs from the outside world. Each buffer is initialized with a given number of data units, and the initial buffer size along a given edge equals the delay along that edge. Additionally, we can model inputs to the system as nodes with no edges leading to them, only edges originating from them. In the case of DMIs, the inputs from the outside world generally exist in the form of sensors. Thus, the nodes in the first level of the DMI graph represent the set of sensors generating data used to control the instrument. We can model

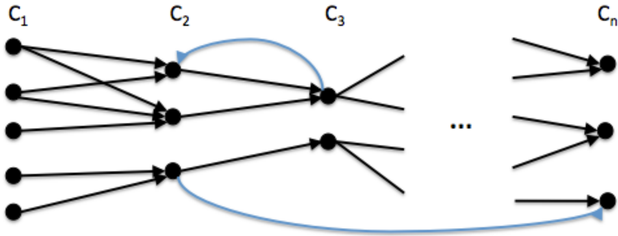


**Figure 4: Data flow model of an arbitrary, non-strictly sequential DMI. The two edges in blue represent the non-sequential pipelines. The upper edge represents a feedback loop, while the other edge represents data that skips layers.**

varying sensor collection rates by labeling each node's output as the collection rate, i.e., the number of samples gathered in one unit of time.

# 4. APPLYING THE MODEL

In this section, we discuss some design decisions facilitated by our graph-theoretic model of the DMI data flow.

## 4.1 Inputs and Outputs

The data flow model exposes relationships between DMI inputs and outputs. To find all outputs that are affected by a given input, we construct a spanning tree of the subgraph affected by the given input node. To do this, we follow forward edges starting at a given control node without revisiting nodes. Symmetrically, to find all inputs that control a given output, we simply follow all backward edges starting at the given synthesis parameter node without revisiting nodes. These algorithms are simple, but they are useful for checking that inputs impact outputs according to the designer's specifications. A tool that implements these algorithms for a given DMI design could be used by DMI designers to analyze and improve their design choices.

The graph representation of the system also facilitates the consideration of integrality and separability in design decisions. The concepts of integrality and separability were first established by Jacob et al [9]. Integral attributes combine perceptually and are hard to differentiate from one another, while separable ones are not. Similarly, integral control dimensions allow the user to move "diagonally" across the input space, while separable control dimensions force the user to move across one dimension at a time. Furthermore, integral control dimensions are most effectively used to control integral attributes [9]. Applying this principle to DMI design, we see that integral control dimensions should control integral synthesis parameters.

We can use the graph to check that integral control dimensions are in fact used to control integral synthesis parameters. To verify that a set $C$ of integral control nodes control a set $S$ of synthesis parameter nodes, we construct the spanning tree of the subgraph affected by data generated at each control $c_i \in C$. Each spanning tree should cover a subset of $S$, and every member of $S$ should appear in at least one spanning tree. This verification can be done manually, or incorporated into design tools that evaluate design decisions and suggest better designs.

## 4.2 Other Design Decisions

Modeling the DMI data flow as a graph facilitates other design decisions. For example, the designer can clearly plan the placement of operations involved in event detection. Placement of event detection can have a large impact on system performance by either preventing or allowing data from being piped through the system unnecessarily. The designer can also analyze the bandwidth consumed by the system by examining the number of edges between two layers of the graph. Meeting bandwidth requirements is particularly important for networked DMI systems and complex DMIs running on systems with limited resources.

The designer can also use the graph to identify channels of control that are completely independent. Each connected component of the graph operates completely independent of every other connected component. In this sense, each connected component can be viewed as a separate DMI. Furthermore, the input controls piped into a particular connected component will have no impact on any other connected component. Such independent components reveal valuable information about the experience of playing the instrument being designed.

# 5. ANALYSIS OF NAIVE SCHEDULING PROTOCOLS

In this section, we analyze two naive scheduling protocols under our graph-theoretic model. With the help of the model, we can analyze performance of these protocols, and identify shortcomings.

## 5.1 Execution by Any New Input

Perhaps the most naive scheduling protocol executes a node whenever it receives a new input along any incoming edge. We can easily represent the basic data flow within this DMI as a layered graph that is not strictly ordered, as explained above. Each node is also labeled with the number of samples required along each incoming edge for a single execution, as well as the number of samples that a single execution yields along each outgoing edge.

If we use a FIFO queue to model data transmission and storage along edges, we run the risk of unbounded queue growth. To prevent unbounded growth, we can use a specialized queue. The queue has a fixed length equal to the number of sample inputs required by the subsequent node along this edge. Whenever a new input arrives, it is placed at the end of the queue and the top sample is discarded. At this time, the node executes using the set of samples stored along each incoming edge.

This protocol fails to guarantee that synthesis events execute according to the order that the physical gestures controlling those events were executed. To see how this fails, consider a DMI with a single input sensor and two output synthesis parameters, as in Figure 5. Also suppose that the computation time for each of nodes 2, 4, and 6 is $c$ units, while the computation time for each of nodes 3, 5, and 7 is $2c$ units. Also suppose node 1 has a data generation rate of 1 sample every $c$ time units. Because data can be piped along the top channel at sample rate, all data sent along this channel will reach the output parameter node 6. However, the bottom channel cannot keep up with the data generation rate. This will result in either an infinitely long buffer along the edge $(1, 3)$ or data being dropped. Additionally, the same data will trigger a sonic result at node 7 $3c$ time units after the same data triggers a sonic result at node 6. It is worth noting that the designer could solve these problems by executing nodes 2, 4, and 6 twice as frequently as 3, 5, and 7. The protocol proposed in Section 6 would make this adjustment for the designer.

## 5.2 Execution by Specific New Input

Another common protocol executes a node whenever the function receives a new input along a particular incoming

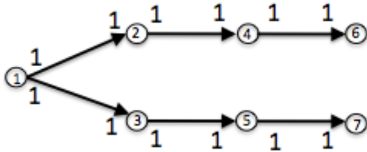


**Figure 5: Arbitrary DMI with a single sensor input and two output synthesis parameters. Each computation both inputs and outputs one unit of data.**

edge. This is the fundamental protocol built into Max/MSP for object triggering. Again, we can represent the basic data flow as a layered graph that is not strictly ordered. Computational input requirements and output results are again depicted as numbers along the edges leading to and from each node.

Just like the previous protocol, this protocol is ill-suited for a classic FIFO queue along edges due to the possibility of unbounded queue growth. Instead, we again use a specialized queue. The queue has a fixed length equal to the number of sample inputs required by the subsequent node along this edge. Whenever a new input arrives, it is placed at the end of the queue and the top sample is discarded. If the new sample arrived along the designated edge, the node executes at this time using the set of samples stored along each incoming edge. This protocol suffers from the same disadvantages as the previous protocol, and an equivalent analysis of the protocol on Figure 5 would demonstrate these problems.

Either of these two protocols can be modified to guarantee desired performance for a specific system. For example, in Figure 5, nodes 3, 5, and 7 might be scheduled to execute half as often. However, the adjustments to the protocols must be tailored to each individual system. Furthermore, depending on the system complexity and configuration, it might be exceedingly difficult to modify these protocols appropriately. We proceed to propose a universally applicable protocol that can be used to automate scheduling of the data flow in any DMI.

# 6. PROPOSED SCHEDULING ALGORITHM

In this section, we propose scheduling the data flow of a DMI according to a periodic admissible sequential schedule (PASS) [12]. A PASS is an ordered list of nodes such that if the nodes are repeatedly executed according to the sequence, all buffers will remain bounded in size.

Lee and Messerschmitt present a class of algorithms, called class $S$ algorithms, that find a PASS for a given data flow if one exists. We propose using the following class $S$ algorithm from [12].

1. Solve for the smallest positive integer vector $q \in \eta(M)$, where $M$ is the topology matrix described above and $\eta(M)$ is the nullspace of $M$.
2. Form an arbitrarily ordered list $L$ of all nodes in the system.
3. For each $\alpha \in L$, schedule $\alpha$ if it is runnable, trying each node once.
4. If each node $\alpha$ has been scheduled $q_\alpha$ times, STOP.
5. If no node in $L$ can be scheduled, indicate a deadlock (an error in the graph).
6. Else, go to 3 and repeat.

The vector $q$ found in the first step of the algorithm tells us how many times each node must execute in one cycle of the PASS. Also, since $q$ is the smallest integer vector in the nullspace of $M$, this algorithm will find the shortest possible PASS, meaning that the total number of nodes that must execute is minimized. Adding up the values in $q$ yields the number of nodes that must execute in the PASS (counting nodes that re-execute).

# 7. DMI PERFORMANCE GUARANTEES

The PASS found by this algorithm (if one exists) guarantees certain properties for our DMI.

## 7.1 Stable Buffers

If we use any PASS to schedule the node execution, we are guaranteed stable buffer sizes [12]. This helps prevent system crashes due to unbounded buffer growth overloading memory. It simultaneously helps prevent increasing misalignment of data that can occur when increasingly old data is used by some subset of nodes with growing incoming queues, while other nodes process data of a fixed age.

## 7.2 Bounded Maximal Latency

A PASS guarantees that sufficient amounts of data are buffered along each incoming edge to a given node whenever that node is scheduled to execute. Consequently, a node can fire immediately when its turn arrives. This means that the algorithm makes the most efficient use of computation time. Time spent during a cycle of the PASS consists of the time it takes for the nodes to execute their computations, plus the data transmission and queueing time. No time is spent idly waiting. We present a theorem summarizing this idea.

THEOREM 1. *When using a PASS to schedule node executions, the maximal latency between any input and output is bounded by the time it takes to execute one cycle of the PASS (ignoring delay lines).*

Because the PASS executes every node at least once, each sensor node executes at least once. And by the end of a cycle of the PASS, no buffer sizes have changed, since $q$ is a vector in the nullspace of $M$. This means that no data is "lost" in the middle of the graph, but is piped through to the end.

The speed at which the PASS executes depends on the number of processors available. If a single processor is used, the class $S$ algorithm we use guarantees that it will find the PASS with the smallest maximal latency, since all other PASSes are multiples of the one this algorithm finds [12]. If multiple processors are used, then the maximal latency can be greatly reduced. Optimizing the multi-processor schedule is combinatorially difficult, and heuristic solutions exist [1, 6, 11]. These algorithms can be applied to our graph-theoretic model if multiple processors are available.

## 7.3 Sequential Control

If we use a class $S$ algorithm, we can guarantee sequential control of the outputs (ignoring delay lines) by postponing execution of the nodes in the final layer of the graph, placing them at the end of the PASS. The last layer of nodes only have incoming edges, since they represent functions that produce sound. Because their outputs are not consumed by other nodes, their execution in the PASS may be postponed until the end of the PASS. We can then order the execution of these nodes according to the earliest control data responsible for their generation.

This ordering guarantees sequential control of sonic events within each period of the PASS. Sequential control across periods of the PASS is guaranteed by the fact that each cycle of the PASS must complete before the next may start. This means that all data collected in a single time period of the PASS will be completely piped through the system before new data enters the system.

## 7.4 Sensor Collection Rates

Nodes with no incoming edges represent relationships with the outside world. In the context of DMI modeling, such nodes represent sensors gathering data on physical movements that control the instrument. These nodes comprise the first layer of the graph. A PASS does not handle the buffering of data sent to these nodes. However, using a PASS does give us useful information about acceptable data collection rates for our sensors.

Consider the PASS found by the class $S$ algorithm above. Each node $i$ must execute $q_i$ times in the PASS. Since sensors are represented as nodes, sensor $j$ must also gather data $q_j$ times in the PASS. If the sensor's data collection rate exceeds this, data will accumulate indefinitely along the buffer leading into it, eventually crashing the system. Data gathered by different sensors will also be increasingly misaligned since the age of data taken off the buffer will increase over time. Conversely, if a sensor gathers fewer than $q_s$ samples each period of the PASS, $s$ will not be able to execute when its turn in the PASS arrives. We present this idea in the following theorem.

THEOREM 2. *$\frac{q_i}{P}$ is the required incoming data rate (data collection rate) for sensor $i$, where $P$ is the period of the PASS (the time for one round of the PASS to execute).*

Note that sensor collection rates can be artificially forced to match this requirement. If we need to reduce the collection rate, we simply discard the appropriate number of incoming samples every period of the PASS. Similarly, if we wish to increase the collection rate, we can perform sample interpolation, inserting the appropriate number of "artificial" values every period of the PASS. These artificial samples might take their value from their "real" neighbors, an average of their neighbors, or some other reasonable estimate of what we would expect the inserted data to be.

# 8. CONCLUSIONS

DMI design is a complex problem, frustrated by varying sensor rates and computational times, data alignment, latency requirements, and buffering problems, among other challenges. In this paper, we propose a graph-theoretic model derived from a digital signal processing model that alleviates these problems by providing a framework within which DMIs can be designed and analyzed. We also provide a scheduling algorithm of functional components within the DMI, and prove that it guarantees several desirable properties for the resulting DMI. This is the first such framework proposed for the design and analysis of DMIs, and serves as a starting point for further development of the framework and scheduling algorithms. The importance of this work will only increase as DMIs become increasingly complex with the introduction of new sensing technologies, mappings, synthesis algorithms, and distributed performance opportunities, and as the bar rises for DMI performance expectations.

# 9. ACKNOWLEDGMENTS

This work was supported by an award from the Microsoft Research Graduate Women's Scholarship Program.

# 10. REFERENCES

[1] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[2] P. Arumi and X. Amatriain. Time-triggered static schedulable dataflows for multimedia systems. In *IS&T/SPIE Electronic Imaging*, 2009.

[3] C. Bartlette, D. Headlam, M. Bocko, and G. Velikic. Effect of network latency on interactive musical performance. *Music Perception*, 24(1):49–62, 2006.

[4] F. Bevilacqua, R. Müller, and N. Schnell. MnM: a Max/MSP mapping toolbox. In *Proc. NIME*, pages 85–88, 2005.

[5] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. IEEE*, 79(9):1305–1320, 1991.

[6] T. Hu. Parallel sequencing and assembly line problems. *Operations research*, 9(6):841–848, 1961.

[7] A. Hunt and M. Wanderley. Mapping performer parameters to synthesis engines. *Organised Sound*, 7(2):97–108, 2002.

[8] A. Hunt, M. Wanderley, and M. Paradis. The importance of parameter mapping in electronic instrument design. *Journal of New Music Research*, 32(4):429–440, 2003.

[9] R. Jacob, L. Sibert, D. McFarlane, and M. Mullen Jr. Integrality and separability of input devices. *ACM TOCHI*, 1(1):3–26, 1994.

[10] R. Karp and R. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[11] W. Kohler. A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. *IEEE Transactions on Computers*, 100(12):1235–1238, 1975.

[12] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 100(1):24–35, 1987.

[13] E. Lee and D. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, 1987.

[14] J. Malloch, S. Sinclair, and M. Wanderley. From controller to sound: Tools for collaborative development of digital musical instruments. In *Proc. ICMC*, pages 65–72, 2007.

[15] J. Malloch, S. Sinclair, and M. Wanderley. A network-based framework for collaborative development and performance of digital musical instruments. *Proc. CMMR*, pages 401–425, 2008.

[16] G. Odowichuk, S. Trail, P. Driessen, W. Nie, and W. Page. Sensor fusion: Towards a fully expressive 3d music control interface. In *IEEE PacRim*, pages 836–841, 2011.

[17] M. Puckette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

[18] A. Schmeder, A. Freed, and D. Wessel. Best practices for Open Sound Control. In *Proc. Linux Audio Conference*, volume 10, 2010.

[19] M. Wanderley. Gestural control of music. In *Human Supervision and Control in Engineering and Music*, pages 632–644, 2001.

[20] M. Wanderley, N. Schnell, and J. Rovan. ESCHER - modeling and performing composed instruments in real-time. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 2, pages 1080–1084, 1998.

[21] J. Wright and E. Brandt. System-level MIDI performance testing. In *Proc. ICMC*, pages 318–321, 2001.