

# Time-triggered Static Schedulable Dataflows for Multimedia Systems

Pau Arumí<sup>a</sup> and Xavier Amatriain<sup>b</sup>

<sup>a</sup>Fundació Barcelona Media - Universitat Pompeu Fabra, C. Ocata, 1, 08003 Barcelona, Spain.

<sup>b</sup>Telefonica Research, Via Augusta, 177, 08021 Barcelona, Spain

## ABSTRACT

Software-based reactive multimedia computation systems are pervasive today in desktops but also in mobile and ultra-portable devices. Most such systems offer a callback-based architecture to incorporate specific stream processing. The Synchronous Dataflow model (SDF) and its variants are appropriate for many continuous stream processing problems such as the ones involving video and audio. SDF allows for static scheduling of multi-rate processing graphs therefore enabling optimal run-time efficiency. But the SDF abstraction does not adapt well to real-time processing because it lacks the notion of time: executing non-trivial schedules of multi-rate dataflows in a time-triggered callback architecture, though possible through buffering, causes jitter, excessive latency and run-time inefficiencies. In this paper we formally describe a new Time-Triggered SDF (TTSDF) model with a static scheduling algorithm that produces periodic schedules than can be split among several callback activations, solving the above-mentioned problems. The model has the same expressiveness than SDF, in the sense that any graph computable by one model will also be computable by the other. Additionally, it enables parallelization and run time load balancing between callback activations.

**Keywords:** Actor-oriented design, Dataflow, Static Scheduling, Callback architecture, Real-time Operating System, Multimedia System Design Methodology, Time-Triggered Synchronous Dataflow

## 1. INTRODUCTION

Operating systems with multimedia processing capabilities are nowadays deployed in all kinds of hardware ranging from desktop workstations to ultra-light portable devices. The high-level services offered by such operating systems added to the fact that all processing is done in software, enables for quick application development. These applications can fulfill the *soft real-time* and low-latency requirements needed in tasks such as real-time video and audio processing. Because of the flexibility and immediacy of these solutions, software-based architectures are often preferred to system-on-chip or embedded software written for a specific device.

Many of the processing algorithms included in these multimedia applications can be naturally expressed as a graph of processing blocks (*actors*) that transforms an infinite stream of data. We are specially interested in models supporting multi-rate behaviour, in that actors may consume different —but fixed— amounts of data, resulting in different rates of execution \*. This is typical for spectral domain processes and for mixed stream types, e.g., video frames, time domain audio frames and spectral domain audio frames.

These kinds of algorithms are very well modeled using *Synchronous Dataflow (SDF)*. The SDF, introduced by Lee and Messerschmitt,<sup>1</sup> is a well studied and used model of computation, in which all actors are processing blocks running *synchronously*, thus the amount of data consumed and produced on each block execution is known *a priori* and does not change. SDF is a *statically schedulable dataflow* and has strong formal properties that makes many questions, like computability and needed buffer sizes, decidable.

But the integration of SDF graphs in a real-time multimedia-enabled operating systems is problematic. In such systems, the real-time processing is done in callback functions, typically triggered by regularly timed

---

Further author information: (Send correspondence to Pau Arumí)

Pau Arumí: E-mail: parumi@iua.upf.edu

Xavier Amatriain: E-mail: xar@tid.es

\*For example, for each execution of a given actor *a*, actor *b* runs two times and actor *c* three times.

hardware interruptions. Though it is possible to adapt the SDF and the outside world by buffering, the SDF model cannot guaranty the absence of jitter, nor operating in an optimum latency or run-time performance.

We propose a new model called *Time-Triggered Synchronous Dataflow (TTSDF)*, which adds two new types of actors and restrictions on the ordering this actors take in the periodic schedule. We show that the new model is equally powerful in the sense that all graphs runnable by the SDF model also run by the TTSDF model. As the name suggest TTSDF naturally fits synchronous dataflows in a callback-triggered scheme. Moreover the model guarantees two important properties: the absence of jitter and the guarantee of an optimum latency.

In the next section we will establish the grounds by describing the *dataflow* semantics and the SDF model of computation. We will then detail the problems that we face when trying to integrate the SDF model in a callback-based architecture in section 3. The TTSDF model is described and proved in section 4. The model is then related with similar work in section 5. Finally we finish in section 6 with some conclusions and future directions.

## 2. BACKGROUND

### 2.1 Dataflows

The term *dataflow* has been and still is often used with loose semantics. We will adhere to the precise definition given by Lee and Parks,<sup>2</sup> who formalized the semantics outlined by Dennis in.<sup>3</sup> The dataflow model of computation consists in a directed graph whose nodes are actors that communicate exclusively through the channels represented by the graph arcs. Conceptually, each channel consist in a *first-in-first-out* (FIFO) queue of tokens. Processing blocks expose how many tokens they need to consume and produce to each channel on the next execution, and they are only allowed to run when tokens stored in the input channels queues are sufficient. The amount of tokens to be consumed or produced by an actor in one execution is also known as *token rate*. As a consequence, processing blocks of a dataflow can be executed in any order (as long as it does not result in negative buffer sizes) and the produced result will not be affected. Although the graph can run sequentially its declarative-style definition makes parallelism explicit and this can be easily exploited.

In *Dynamic Dataflows* (DDF), where token rates are allowed to change, a run-time scheduling director is needed. The question if an arbitrary dataflow can run with finite buffers size is a non-decidable problem.<sup>2</sup> The *Process Networks (PN)* model<sup>45</sup> is very much related to dataflows. Process Networks, however, do not need to expose the token rate of their processing blocks. Each actor runs in a separate process which is orchestrated by a scheduler similar to those found in general purpose operating systems. Like DDF's, PN's computability is non-decidable. Unfortunately, in the general case it is impossible to say anything about the maximum latency of the system, therefore it is not a good match to real-time requirements.

### 2.2 Synchronous Dataflow (SDF)

An SDF is a dataflow model in which token rates are known *a priori* and can not change during run-time. This might be a limitation, however more dynamic and complex system can be achieved by composing SDF subsystems using Process Networks or Dynamic Dataflow models. This fixed-rate restriction limits the model expressiveness but, on the other hand, many questions are decidable, like its calculability and needed buffers. The periodic scheduling can be found *statically*, before run-time; therefore all the scheduling overhead evaporates. Furthermore, it enables optimized embedded software synthesis.<sup>6</sup>

The Dataflow models and the SDF in particular are widely used in multimedia processing systems, although in most of the cases the multi-rate capabilities are not fully exploited. To name a few of them: Max/MSP, PureData, GStreamer or Jack (an inter-application router for audio and, recently, video).

#### 2.2.1 Static scheduling of SDF graphs

A *static schedule* consists of a finite list of actors, in the case of a sequential schedule, and  $N$  finite lists —with additional synchronization information— in the case of a  $N$  parallel schedule <sup>†</sup>. In any case, the schedule is

---

<sup>†</sup>To obtain optimal parallel schedules, the amount of work for each actor execution should be fixed and known. Note that static schedulings have very little in common with OS shedding algorithms such as *rate-monotonic scheduling* or *earliest deadline first scheduling*

*periodic*. An important property of an admissible schedule is that the buffering state (that is, the size of each FIFO queue) after a whole period has been executed exactly matches the initial state.

The static scheduler algorithm have two phases: the first one consists on finding how many times each actor must run in one period, and the second one consists on simulating an execution of a period. The next paragraphs detail these two phases taking the graph in Figure 1 as example.

As noted above, buffer sizes must reach the initial state after executing a whole period. As a consequence we can write that, within a cycle, the total number of tokens produced into any queue must balance the total number of tokens consumed from that queue; and this is precisely captured by the *balance equations*. It is proved<sup>1</sup> that if a non-zero solution of the balance equations exists, an integer solution also exists (and not only one but an infinite number of them). Solving the balance equations is equivalent to finding a vector in the *null-space* of the graph *topology matrix*. The topology matrix is similar to the incidence matrix in graph theory and is constructed as follows: all node and arcs are first enumerated, the  $(i, j)$ th entry in the matrix is the amount of data produced by node  $j$  on arc  $i$  each time it is invoked. If node  $j$  consumes data from arc  $i$ , the number is negative, and if it is a connected to arc  $i$ , the number is zero. Therefore we assign a column to each node and a row to each arc.

The second phase of the static scheduling algorithm consists on simulating a cycle execution. Iteratively, all nodes are checked for their runnability (that is, have enough tokens in their inputs), and, if runnable, they are scheduled. This goes on until the number of executions  $\vec{q}$  found in the previous phase are completed. It is proved<sup>1</sup> that if an admissible schedule exists, this straightforward strategy will find such a schedule. It is interesting to note that any rate-consistent graph will have a schedule unless the graph has loops, and there is a lack of initial delays in the looping arcs.

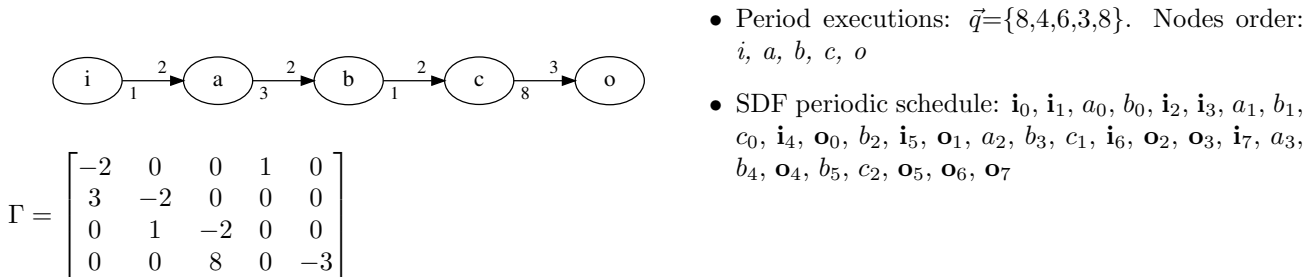


Figure 1: A simple SDF graph, its topology matrix and its non-trivial scheduling

### 3. SDF IN TIMED ARCHITECTURES

An SDF schedule, like the one in the previous example, does not fit well in timed callback-based architecture. On one hand, in such architectures, callbacks with external inputs are triggered in regular timings and outputs are to be produced within deadlines. On the other hand, the SDF model is an *untimed* abstraction because timing constrains are not captured by the model.

Ideally we would like to combine these two worlds: *timed* executions of some dataflow actors —the ones linked with callback inputs and outputs— followed by sequences of untimed actors. But, since a periodic SDF scheduling may contain many instances of each, timed or untimed, actor, such combination is not obvious.

A different, and simpler, approach consists on doing buffering and blocking reads/writes. But this approach is flawed and not usable in the general case. It is easy to find examples of this in audio (programming) libraries that offer access to the audio device, such as the Linux Alsa, and the cross-platform Portaudio.<sup>7</sup> Such *callback to blocking interface adaptation* is implemented with a user’s processing thread that synchronises with the callback thread, exclusively dedicated to buffer management. However this adaptation does not work well for multi-rate dataflows. To illustrate the exact problems we will use the example of SDF scheduling in Figure 1. The chronogram in Figure 3 shows the buffering and blocking operations during its run-time. We will assume that the callback real-time deadlines are always met.

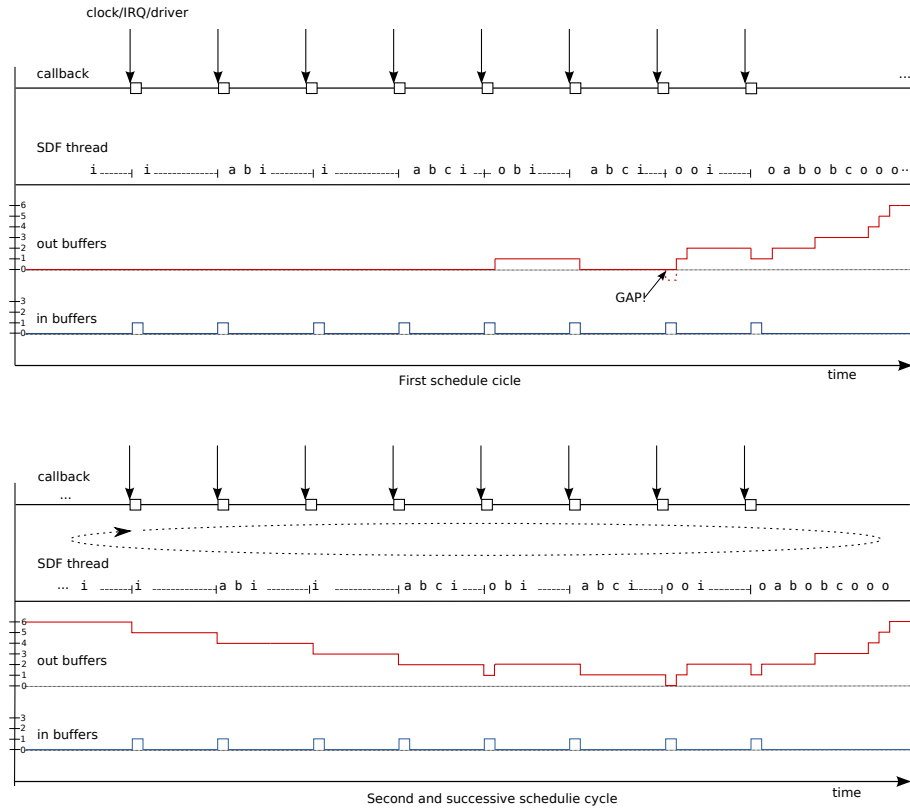


Figure 2: A sequence diagram (chronogram) showing the problematic adaptation of an SDF schedule (from Figure 1) on top of a callback architecture. The line on the top shows the callback process that deals with input and output buffering. The second line shows the SDF execution thread, with blocking executions of actor **i** (input). Next two lines show the size of output and input buffers. Points marked with *GAP!* indicate a lack of output buffer to be served by the callback. This will produce a gap in the output stream.

A detailed analysis of the chronogram shows several problems of running an SDF schedule in a callback-based architecture: **a)** It introduces more latency than necessary. The execution order could have been arranged in a different way so that the first output would be produced several callbacks earlier. **b)** It introduces *jitter* — that is, the latency is not stable but increments causing a *gap*—, at least during the first cycle execution, and possibly in successive cycles depending on how buffering is managed. **c)** It introduces run-time overhead because of the context switching of threads. Moreover in some architectures (e.g., some audio plug-in architectures) spawning threads is not allowed. In other cases it is allowed, but they run at lower priority than the callback process.

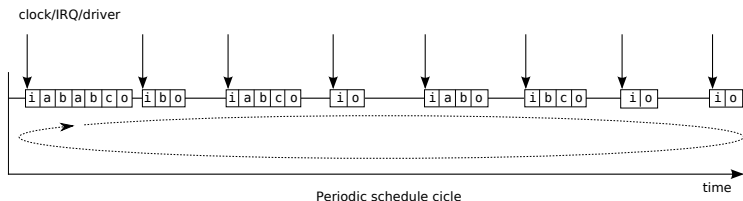


Figure 3: The desired way of running the same SDF periodic schedule (defined in Figure 1). Now processing blocks are run inside the callback and no buffering is required nor blocking reads/writes. The reason why this is possible is that inputs and outputs are well distributed. However, before running the periodic schedule, the following prologue is needed to initialize the internal dataflow queues:  $(i,o)$ ,  $(i,o)$ ,  $(i,o)$

Therefore we need the new approach outlined above: combine regularly triggered actors that consume the external inputs with other untimed actors. We would also like to enable the scheduling to run *inside* the callback process, avoiding jitter and guaranteeing optimum latency. Such a wish list is realized in the chronogram in Figure 3, for the same SDF example. Although the example in Figure 3 shows a sequential computation—all actors are executed in the callback process—it is still possible to optimize the run-time by parallelizing on multiple processors in the same way that is done in SDF graphs. Now that the problem and a possible solution have been illustrated we are ready to precisely define a new model of computation that suites that purpose.

#### 4. TIME-TRIGGERED SYNCHRONOUS DATAFLOW (TTSDF)

Our goal is to find a systematic methodology or algorithm for obtaining sequential schedules of actors in a way that can be run in the real-time callback function, avoiding jitter and optimizing the latency. Additionally we want to prove that all graphs that have an SDF schedule also have a schedule that fulfils these conditions. This work follows a similar development than the one found in,<sup>1</sup> on statical scheduling of synchronous dataflows and builds upon some of its definitions, lemmas, and theorems.

DEFINITION 1. A Time-Triggered Synchronous Data Flow (TTSDF) is a connected and directed graph with each arc labeled with two integers, one corresponding to the amount of samples being produced to that arc and the other corresponding to the amount of samples being consumed from that arc, a subset of source nodes  $I$  or inputs, a subset of sink nodes  $O$  or outputs, and  $\vec{b}(0)$  a vector containing the initial buffer sizes of all arcs.

Note that all input nodes are sources and all output nodes are sinks but the opposite is not true. The only difference with an SDF graph is that a TTSDF comes with some sources tagged as inputs and some sinks tagged as outputs. Like in SDF, the number of tokens consumed and produced for each block execution is known *a priori*. Note that any SDF model can be interpreted as a TTSDF model with no sink or source tagged as input or output.

##### 4.1 The TTSDF computation model

Dataflow models, and SDF and TTSDF in particular, are *coordination languages* as defined in<sup>8</sup> and.<sup>2</sup> Coordination languages permit nodes in a graph to contain arbitrary subprograms, but define a precise semantic for the interaction between nodes. We call *host languages* those languages used to define the subprograms, which are usually conventional languages such as C or C++.

We give the semantics of the TTSDF coordination language by defining the interaction between time-triggered callbacks, timed (input/output) actors and untimed actors. Furthermore, we use a mathematical computation model to reason about the admissible schedules of the dataflow.

###### 4.1.1 Callback-based Coordination Language

Three types of actors are distinguished: *inputs*, *outputs* and *untimed* actors. Inputs are time-triggered, that is, their execution is driven by the time-triggered callback. Each input is associated to a callback buffer, each of them may consist in different number of tokens. All inputs run together sequentially in an arbitrary order. An input execution consists on reading its callback input buffer and sending out tokens to its dataflow queues. Immediately after the inputs, zero or more untimed nodes are run. Note that untimed actors can be either filters, sources or sinks.

Outputs closes the execution sequence triggered by the callback. As the inputs, they run together in an arbitrary order. Termination of outputs execution finishes the callback function. Conversely to inputs, each output writes tokens from its queues into its callback output buffer. Outputs are not time-triggered but are time-restricted because they must run before the deadline (the next callback)

Each execution sequence triggered by a callback is called *callback activation*. The dataflow semantics are preserved by an important restriction: the concatenation of one or more callbacks activations must form a periodic scheduling.

A first consequence of this model is that jitter can not exist because no input or output buffer can be missed. When a callback activation finishes, the outputs are always filled with new data<sup>‡</sup>. Finally note that we always refer to the "callback function", but a "function" is not technically necessary. It could also be a similar —and equivalent— scheme like a thread that is awoken by a time-trigger. However, defining callback functions is the typical solution used in open architectures to allow developers to plug in their processing algorithm.

#### 4.1.2 Formal Computation Model

DEFINITION 2.  $\wp(A)$  indicates an arbitrary sequence of all elements in the set  $A$ .

DEFINITION 3. A **callback activation** is any a sequence in the following language:

$C = \{\wp(I)(V - (I \cup O)^*)\wp(O)\}$ , and  $l$  callback activations is any sequence in  $C^l$ , where  $V$  is the set of nodes in the graph, and  $I$  and  $O$  are the nodes labeled as inputs and outputs respectively.

DEFINITION 4. A **callback order** is any prefix of the language  $C^*$

The following sequences  $A$  and  $B$  are examples of a callback order —and all its sub-sequences as well— while  $C$  is not in callback order. Assume that the inputs and outputs are  $I = \{i_a, i_b, i_c\}$ ,  $O = \{o_d, o_e\}$  and the rest of nodes  $V - (I \cup O) = \{m, n\}$ . Parenthesis are used to indicate complete callback activations

$$\begin{aligned} A &= [(i_a, i_b, i_c, m, n, o_d, o_e), (i_a, i_b, i_c, n, n, o_d, o_e)] \\ B &= [(i_a, i_b, i_c, o_d, o_e), i_a, i_b] \\ C &= [i_a, i_b, m, i_a] \end{aligned} \quad (1)$$

As the SDF, this computational model consists on a graph with a FIFO queue on each arc that gets tokens from its producer processing node and passes them on to its consumer processing node. Such queues will be called *buffers* and their size after  $n$  executions (or firings) can be computed as follows:  $\vec{b}(n+1) = \vec{b}(n)\Gamma\vec{v}(n)$ ; where  $\vec{v}(n)$  represents the processing node being executed in  $n$ th place. Each  $\vec{v}(n)$  is a vector with a 1 at the position corresponding to the node to be executed and zeros all the rest. Taking the simple graph in Figure 4 as example, it needs two executions of nodes 1 and 3 and one execution of node 2 to complete a period. This is the execution rate per period and is given by the vector  $\vec{q}$  which is the smallest integer vector in the *nullspace* of the graph topology matrix  $\Gamma$ :

$$\Gamma = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 1 & -2 \end{bmatrix} \Gamma\vec{q} = \vec{0}; \vec{q} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad (2)$$

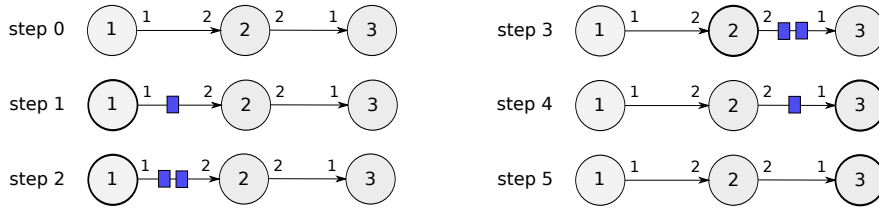


Figure 4: A sequence of executions in a simple graph. Nodes in bold line represent the last fired node, which correspond to vector  $\vec{v}(n)$ . Tokens in the two arcs correspond to vector  $\vec{b}(n)$

An execution is valid if it fulfils two conditions: the buffer size  $\vec{b}(n)$  remains a non-negatives vector, and the sequence of firings remains in *callback order*. This second condition is what makes TTSDF a different model than SDF. This condition means that any allowed sequence of execution begins with all inputs, then any number of non input/outputs, then all the outputs, and then it starts with the inputs again. Note that non-input sources and

<sup>‡</sup>which can either be a transformation of the input buffers provided in the same callback or a transformation of inputs in previous callbacks.

non-output sinks can exist and, regarding the callback order, are considered exactly as other non input/output nodes. In our example, assuming that node 1 is an input and node 3 an output, Figure 4 shows a valid TTSDF scheduling.

## 4.2 Static scheduling of TTSDF graphs

DEFINITION 5. (From<sup>1</sup>) A **periodic admissible sequential schedule (PASS)** is a sequence including all nodes in the graph such that if the nodes are executed in that order the amount of tokens in the buffers (or buffer sizes) will remain non-negative and bounded.

DEFINITION 6. A **callback periodic admissible sequential schedule (CPASS)** is a periodic and infinite admissible sequential schedule in callback order. It is specified by a list  $\phi$  that represents one execution period. The period of a CPASS is the number of nodes in one execution period.

DEFINITION 7. An **l-latency CPASS** is an infinite admissible sequential schedule in callback order consisting in two parts  $(\sigma, \phi)$ : The first part is a finite prologue  $\sigma$  made of  $l$  empty callback activations, that is  $(\wp(I)\wp(O))^l$ . The second part is an infinite schedule specified by one execution period  $\phi$ .

The scheduled TTSDF graph is a modified version of the given process graph in which buffers (delays) are added to the incoming arcs of the output nodes. The amount of added delay tokens to an output arc is  $rl$  where  $r$  is the consuming rate of the arc and  $l$  the given latency.

This means that, in terms of communication with the outside world, the first  $l$  callback activations (the prologue) will only buffer in new tokens and buffer out the added delays. Starting from the  $(l + 1)$ th execution the output nodes will produce tokens that have actually been processed by the graph. In typical situations these correspond to transformations of tokens from the input nodes. Adding delays enables the schedule to do buffering while respecting the callback order restriction. As we will see, this buffering is necessary in order to reach a (buffering) state in which a periodic execution schedule in callback order exists.

THEOREM 1. **Necessary conditions** Given a TTSDF with topology matrix  $\Gamma$ , the two following conditions are necessary for the existence of a CPASS: First,  $\text{rank}(\Gamma) = s - 1$ , where  $s$  is the number of nodes; and second, for any non-zero  $\vec{q}$  such that  $\Gamma\vec{q} = \vec{0}$  it is true that  $\forall i \in I \cup O, \vec{q}_i = r$  for some integer  $r$

In other words, if these two conditions are not fulfilled a CPASS will not exist. Compared to the SDF model the TTSDF just adds the second condition. This condition means that all nodes marked as input and output must have the same rate in terms of number of executions per period. We will now prove the theorem validity.

PROOF 1. The proof for the first condition can be found in<sup>1</sup> (it is valid for both SDF and TTSDF). The second condition will be proved by contraposition: we have to see that if two input/output nodes are to be executed a different number of times per cycle then a CPASS does not exist. Let be  $i$  and  $j$  indices such that  $\vec{q}_i > \vec{q}_j$  and let  $n = \vec{q}_i - \vec{q}_j$ . After running  $m$  complete periods node  $i$  will run exactly  $mn$  more times than node  $j$ . A CPASS, on the other hand, imposes that after each cycle terminates, all input/output nodes have been executed the same number of times. So a CPASS does not exist for such  $\vec{q}$ . Q.E.D.

Our algorithm for finding a CPASS will begin by checking these two necessary conditions. We now need a sufficient condition that asserts that a CPASS exists for a given TTSDF graph. To that purpose we will characterize a class of algorithms and prove that if a CPASS exists the algorithm will find it, and return failure if not.

DEFINITION 8. **Class C algorithms** (“C” for callback) : An  $i$ th node is said to be runnable at a given position if it has not been run  $\vec{q}_i$  times and running it will not cause any buffer size to go negative. Given a latency  $l$ , a positive integer vector  $\vec{q}$  such that  $\Gamma\vec{q} = \vec{0}$  and an initial state for the buffers  $\vec{b}(0)$ , a **class C algorithm** is any two steps algorithm that: First, initialises  $\vec{b}(0)$  adding delays to the incoming arcs of output nodes (as many delays as  $l$  executions of the outputs will consume), defines the prologue  $\sigma$  as any sequence in  $(\wp(I)\wp(O))^l$ , and updates  $\vec{b}$  accordingly. And second, schedules the period  $\phi$  with the given  $\vec{q}$ : schedules any node if is runnable and does not break the callback order, and updates  $\vec{b}$ , and stops only when no more nodes are runnable. If the periodic schedule terminates before it has scheduled each node  $i$   $\vec{q}_i$  times it is said to be deadlocked, else it terminates successfully.

LEMMA 1. *To determine whether a node  $x$  in an SDF graph can be scheduled at time  $i$ , it is sufficient to know how many times  $x$  and its predecessors have been scheduled, and to know  $b(0)$ , the initial state of the buffers. That is, we need not know in what order the predecessors were scheduled nor what other nodes have been scheduled in between.*

This lemma is quite intuitive and it is formally proved in,<sup>1</sup> thus we skip its proof.

LEMMA 2. **Sequence permutations are in callback-activation phase:** *Two sequences  $\phi$  and  $\chi$  in callback order such that one is a permutation of the other, either both ends with a node in  $V - (I \cup O)$  (that is, not necessarily the same) or both ends with the same node. And so, with the same prefix of  $\wp(I)$  or  $\wp(O)$*

PROOF 2. *By contraposition: Assume that a sequence, say  $\phi$  ends with a  $\wp(I)$  or  $\wp(O)$  sub-sequence and  $\chi$  ends with a node different than the last node in  $\phi$ . Then, the ending  $\wp(I)$  or  $\wp(O)$  sub-sequence of  $\phi$  is different than the ending sequence of  $\chi$ . But incomplete sub-sequences of  $\wp(I)$  and  $\wp(O)$  are only allowed at the end of the sequence, by callback order definition. Therefore,  $\phi$  and  $\chi$  differ in the total number of nodes in  $I$  or  $O$ , and they can not be permutations. Q.E.D*

THEOREM 2. **Sufficient condition:** *Given a TTSDF with topology matrix  $\Gamma$  such that  $\text{rank}(\Gamma) = s - 1$  and given a positive integer vector  $\vec{q}$  such that  $\Gamma\vec{q} = 0$ , and all input/output node  $i$  have equal  $\vec{q}_i$  (the "necessary conditions"); If a latency- $l$  CPASS exists of period  $p = \vec{1}^T \vec{q}$  exists, any class C algorithm will find such a CPASS. Successful completion of these algorithms is a sufficient condition for the existence of the CPASS.*

PROOF 3. *It is sufficient to prove that if an  $l$ -latency TTSDF scheduling  $(\sigma, \phi)$  of latency  $l$  and period  $p$  exists, a class C algorithm will find such scheduling. That is, it will not deadlock before the termination condition is satisfied. Trivially, any class C algorithm will find a prologue equivalent to  $\sigma$ , since the different permutations  $\wp(I)$  and  $\wp(O)$  do not affect the runnability of further nodes. Let's now demonstrate that such algorithm will find the periodic part of the scheduling. We need to show that if the algorithm schedules  $\chi(n)$  for the first  $n$  executions, where  $0 \leq n < p$ , it does not deadlock for its  $(n + 1)$ th execution before  $n = p$ .*

*Lets proceed doing a case analysis of the  $n$ th element in  $\chi(n)$  (that is the last element scheduled). If the  $n$ th element in  $\chi(n)$  is an input but its  $\wp(I)$  sub-sequence has not been completed then the next input in  $\wp(I)$  will be scheduled. If the  $n$ th element in  $\chi(n)$  is an output but its  $\wp(O)$  sub-sequence has not been completed then the next output in  $\wp(O)$  will be scheduled. If the  $n$ th element in  $\chi(n)$  is an output that closes a complete  $\wp(O)$  sequence, the first element in  $\wp(I)$  will be scheduled, because the callback order of  $\chi(n)$  guarantees that, given that  $n < p$ , at least another callback activation can be scheduled.*

*The remaining cases —the non trivial ones— are that the  $n$ th element scheduled is either the last of a  $\wp(I)$  sub-sequence or is a node in  $V - (I \cup O)$  (a non input/output). Since an  $l$ -latency CPASS exists, assume that  $\phi(n)$  are the  $n$  first entries of the periodic part of such CPASS.*

*If  $\chi(n)$  is a permutation of  $\phi(n)$  then the  $(n + 1)$ th element in  $\phi$  is runnable by Lemma 1 and 2. Lemma 1 tells us that the runnability of a node depends on how many times its predecessors have run, but not their order. And Lemma 2 says that permutations end at the same part of a callback activation, and so the  $(n + 1)$ th element in  $\phi$  will not break the callback order.*

*If  $\chi(n)$  is not a permutation of  $\phi(n)$  then at least one node appears more times in  $\phi(n)$  than in  $\chi(n)$ . Let  $\alpha$  be the first such node.*

*We can prove that  $\alpha \notin I$ . This can be seen by contraposition: Let's assume that  $\alpha \in I$ . Let  $j$  be the position that  $\alpha$  takes in  $\phi$ .  $\phi(j - 1)$  and  $\chi(j - 1)$  are clearly permutations. Since  $\phi$  is in callback order,  $\alpha$  is preceded by  $\wp(O)$  and zero or more inputs. But Lemma 2 states that two permutations in callback order must end at the same phase of the callback activation. That means that the whole  $\wp(O)\wp(I)$  sub-sequence will remain in permutation order, and  $\alpha$  can not be the first different node.*

*In the other cases,  $\alpha \in O$  or  $\alpha \in V - (I \cup O)$ ,  $\alpha$  can be scheduled as the  $(n + 1)$ th element in  $\phi$  because it is runnable by Lemma 1 and it will not break the callback order. Q.E.D*

THEOREM 3. **TTSDF and SDF have equivalent computability:** *If a TTSDF has a PASS, then it also has an  $l$ -latency CPASS for some  $l \leq p$ .*

PROOF 4. We can prove it by constructing a  $p$ -latency CPASS out of a PASS. Let  $\Gamma$  be the topology matrix of the graph, and assume that the PASS is characterised by a period of executions  $\phi$ , and has been calculated with a given  $\vec{q}$  such that  $\Gamma\vec{q} = \vec{0}$  and has period  $p = \vec{1}^T \vec{q}$ .

First, note that on any synchronous dataflow scheduling source nodes can be pushed towards the beginning and sink nodes can be pushed toward the end. Such modifications of the scheduling sequence result in increased buffer sizes, but they do not change the runnability of any successor node. Also note that we can transform a periodic schedule  $\phi$  into a prologue and periodic part, by spanning two periods and slicing it with the same period size. To be precise, for any given  $n$  the new prologue is the  $n$  first elements in  $\phi$ , and the new period is the last elements in  $\phi$  starting from the  $n + 1$ th concatenated, again, with the first  $n$  elements in  $\phi$ .

With the previous observations, we will now show how to transform a PASS into a  $p$ -latency CPASS. **1)** Push all input nodes in  $\phi$  to the beginning, forming sub-sequences of  $\wp(I)$ . And push all output nodes to the end, forming sub-sequences of  $\wp(O)$ .  $\phi' = (\wp(I)\wp(I)\dots\wp(O)\wp(O))$  **2)** Let  $n$  be the number of  $\wp(I)$ s in  $\phi'$ . We define the prologue  $\sigma$  as  $\wp(I)^{n-1}$ , and the new period  $\phi''$  as  $\phi'$  taking out the prologue and concatenating it to the end, applying the aforementioned technique of slicing two spanned periods. **3)** Modify  $\phi''$  by pushing the  $n - 1$   $\wp(I)$  sub-sequences towards the beginning, next to each  $\wp(O)$ . **4)** Add  $n - 1$   $\wp(O)$ s at the prologue which will be executed with the added delays of the CPASS. Q.E.D.

COROLARY 1. We can choose the smallest  $\vec{q}$  in the null-space of  $\Gamma$  to find the schedule.

PROOF 5. We can choose the smallest  $\vec{q}$  for finding a PASS as proved in.<sup>1</sup> Theorem 3 (equivalent computability) guarantees that with the same  $\vec{q}$  a CPASS exist.

### 4.3 The Scheduling Algorithm

Given these theorems, we now propose a scheduling algorithm that clearly falls into the defined *class C algorithms*, and therefore, will find an  $l$ -latency CPASS if one exists. The algorithm takes the following inputs: Topology matrix  $\Gamma$ , number of nodes  $s$ , added latency  $l$ , initial delays  $\vec{b}_0$

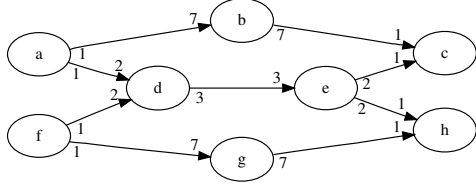
```
def csdf_schedule :
# necessary conditions :
if rank( $\Gamma$ ) != s-1:
return "Rate_mismatch"
 $\vec{q}$  = smallest_integer_in_null-space( $\Gamma$ )
I = arbitrary list of input nodes
O = arbitrary list of output nodes
L = arbit. list of non input/output nodes
for n in O+I : if  $\vec{q}(n) \neq \vec{q}(O_0)$  :
return "In/outs_should_have_same_rate"
 $\vec{v}_{inputs}$  = sum( $\vec{v}_i$  for each i in I)
 $\vec{v}_{outputs}$  = sum( $\vec{v}_o$  for each o in O)
prologue_schd = (IO)l
# set added latency buffering
 $\vec{b} = \vec{b}_0 + \Gamma * \vec{v}_{inputs} * l$ 
 $\vec{x} = \vec{0}$  # the current number of executions
activation_closed=True
while  $\vec{x} \neq \vec{q}$  :
found_any_runnable = False
if activation_closed:
cycle_schd += I
 $\vec{b} += \Gamma * \vec{v}_{inputs}$ 
activation_closed = False
for n in L:
# run node n if runnable
```

```
if min( $\vec{b} + \Gamma * \vec{v}(n)$ )>=0 and  $\vec{x}(n) < \vec{q}(n)$  :
cycle_schd += [n]
 $\vec{x}(n) += 1$ 
 $\vec{b} += \Gamma * \vec{v}(n)$ 
found_runnable = True
# run all output nodes if runnable
if min( $\vec{b} + \Gamma * \vec{v}_{outputs}$ )>=0 and  $\vec{x}(O_0) < \vec{q}(O_0)$  :
cycle_schd += O
for o in O :  $\vec{x}(o) += 1$ 
 $\vec{x} += \Gamma * \vec{v}_{outputs}$ 
found_runnable = True
activation_closed = True
if not found_runnable :
#lacks initial buffering
return "DEADLOCK"
return prologue_schd , cycle_schd
```

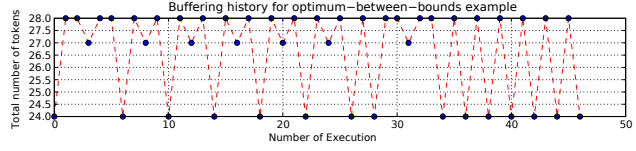
Using the previous function it is easy to find the CPASS with minimum latency:

```
for l in [0, period] :
result = csdf_schedule(l)
if result is not deadlock:
return result
return "DEADLOCK"
```

We show the result of applying the scheduling algorithm in the two examples in figures 5 and 6.



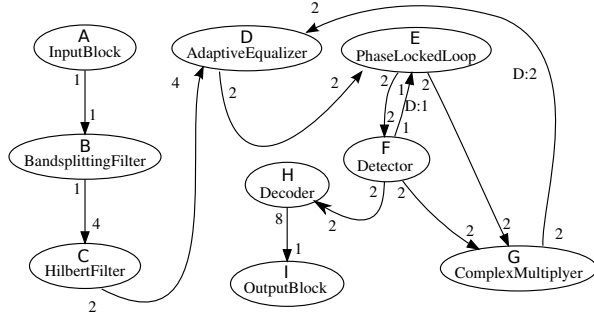
(a) The “optimum” example graph



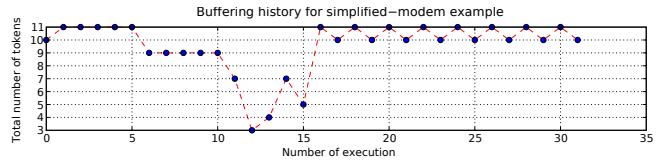
(b) Buffering history of the periodic schedule

- Executions per period  $\vec{q}=\{14, 2, 14, 7, 7, 14, 2, 14\}$ , corresponding to nodes:  $a, b, c, d, e, f, g, h$ .
- Callback-triggered scheduling:  $(\mathbf{a}_0, \mathbf{f}_0, \mathbf{c}_0, \mathbf{h}_0), (\mathbf{a}_1, \mathbf{f}_1, \mathbf{c}_1, \mathbf{h}_1), (\mathbf{a}_2, \mathbf{f}_2, \mathbf{c}_2, \mathbf{h}_2), (\mathbf{a}_3, \mathbf{f}_3, \mathbf{c}_3, \mathbf{h}_3), (\mathbf{a}_4, \mathbf{f}_4, \mathbf{c}_4, \mathbf{h}_4), (\mathbf{a}_5, \mathbf{f}_5, \mathbf{c}_5, \mathbf{h}_5) + (\mathbf{a}_0, \mathbf{f}_0, \mathbf{b}_0, \mathbf{d}_0, \mathbf{e}_0, \mathbf{g}_0, \mathbf{c}_0, \mathbf{h}_0), (\mathbf{a}_1, \mathbf{f}_1, \mathbf{d}_1, \mathbf{e}_1, \mathbf{c}_1, \mathbf{h}_1), (\mathbf{a}_2, \mathbf{f}_2, \mathbf{d}_2, \mathbf{e}_2, \mathbf{c}_2, \mathbf{h}_2), (\mathbf{a}_3, \mathbf{f}_3, \mathbf{d}_3, \mathbf{e}_3, \mathbf{c}_3, \mathbf{h}_3), (\mathbf{a}_4, \mathbf{f}_4, \mathbf{d}_4, \mathbf{e}_4, \mathbf{c}_4, \mathbf{h}_4), (\mathbf{a}_5, \mathbf{f}_5, \mathbf{d}_5, \mathbf{e}_5, \mathbf{c}_5, \mathbf{h}_5), (\mathbf{a}_6, \mathbf{f}_6, \mathbf{c}_6, \mathbf{h}_6), (\mathbf{a}_7, \mathbf{f}_7, \mathbf{b}_1, \mathbf{d}_6, \mathbf{e}_6, \mathbf{g}_1, \mathbf{c}_7, \mathbf{h}_7), (\mathbf{a}_8, \mathbf{f}_8, \mathbf{c}_8, \mathbf{h}_8), (\mathbf{a}_9, \mathbf{f}_9, \mathbf{c}_9, \mathbf{h}_9), (\mathbf{a}_{10}, \mathbf{f}_{10}, \mathbf{c}_{10}, \mathbf{h}_{10}), (\mathbf{a}_{11}, \mathbf{f}_{11}, \mathbf{c}_{11}, \mathbf{h}_{11}), (\mathbf{a}_{12}, \mathbf{f}_{12}, \mathbf{c}_{12}, \mathbf{h}_{12}), (\mathbf{a}_{13}, \mathbf{f}_{13}, \mathbf{c}_{13}, \mathbf{h}_{13})$
- Added latency: 6 external buffers.

Figure 5: Graph “with optimum between bounds”. Example taken from.<sup>9</sup> Callback activations are grouped with parenthesis and prologue and periodic part are separated with a “+”.



(a) A modem example graph



(b) Buffering history of the periodic schedule

- Executions per period  $\vec{q}=\{1, 8, 1, 1, 1, 2, 8, 8, 1\}$  corresponding to nodes:  $D, B, G, H, F, C, A, I, E$ .
- Callback-triggered scheduling:  $(\mathbf{A}_0, \mathbf{I}_0), (\mathbf{A}_1, \mathbf{I}_1), (\mathbf{A}_2, \mathbf{I}_2), (\mathbf{A}_3, \mathbf{I}_3), (\mathbf{A}_4, \mathbf{I}_4), (\mathbf{A}_5, \mathbf{I}_5), (\mathbf{A}_6, \mathbf{I}_6) + (\mathbf{A}_0, \mathbf{B}_0, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3, \mathbf{C}_0, \mathbf{B}_4, \mathbf{B}_5, \mathbf{B}_6, \mathbf{B}_7, \mathbf{C}_1, \mathbf{D}_0, \mathbf{E}_0, \mathbf{F}_0, \mathbf{G}_0, \mathbf{H}_0, \mathbf{I}_0), (\mathbf{A}_1, \mathbf{I}_1), (\mathbf{A}_2, \mathbf{I}_2), (\mathbf{A}_3, \mathbf{I}_3), (\mathbf{A}_4, \mathbf{I}_4), (\mathbf{A}_5, \mathbf{I}_5), (\mathbf{A}_6, \mathbf{I}_6), (\mathbf{A}_7, \mathbf{I}_7)$
- Added latency: 7 external buffers.

Figure 6: An example of modem. Callback activations are grouped with parenthesis and prologue and periodic part are separated with a “+”.

## 5. RELATED WORK

Some extensions of the SDF model exist. Some use a less constrained model allowing rate parameters to change during run-time —such as Boolean-controlled Dataflow<sup>10</sup> and Heterochronos models<sup>11</sup>—; and few others, like TTSDf, introduce the concept of time.

In the timeliness extensions category we find SDF-for-VLSI (VSDF).<sup>12</sup> The VSDF model eases the transition between a synchronous dataflow and an implementation using specialized VLSI synchronous circuits. It introduces a time-aware notation that allows to specify a model and statically verify correct synchronization at Register-transfer Level. Like DT, VLSI is not suited for callback-based systems either.

Other time-related extensions to SDF have been designed to improve the parallelization into multiprocessors using accurate predictions of the execution times for dynamic work-loads.<sup>13</sup> However, this approach is very problem-specific —the presented case models a MPEG-4 video shape-texture decoding system— and it is restricted to homogeneous SDF’s, that is: all port rates are one, and actors runs at the same rate.

Other models and architectures match more closely the concepts of callback and interrupt that are the base for our TTSDf model. Giotto<sup>14</sup> and Simulink with Real-Time Workshop from MathWorks, define and implement

time-triggered models that allow the combination of fast running components with slow running components. Although these time-triggered models can specify different execution rates for their components they do not have dataflow semantics because they lack queues on each arc, and so, execution order independence. Another related model is Discrete Events model.<sup>15</sup> In DE components interact with one another via events that are placed on a time line. We find an interesting example of DE in ChuckK,<sup>16</sup> a concurrent programming language for multimedia.

The Discrete-Event Runtime Framework<sup>17</sup> implemented in the Ptolemy II framework<sup>18</sup> combines time-triggered models and architectures with Discrete Events on one hand, and dataflow untimed computation on the other hand. Therefore a time-triggered actor execution can activate further down-stream executions. Although similar, there is an essential difference with our model. In DE Runtime Framework, the only time-triggered execution corresponds to all inputs executing at the same instant and it finishes with execution of all outputs. In consequence, the down-stream execution is much more restricted than if no outputs needs to be executed. Therefore, the concepts of partitioning the scheduling cycle into callback activations and providing an optimum initial latency to avoid jitter are novel.

## 6. CONCLUSIONS

Existing dataflow models deal with multi-rate processing. However, they are not adequate for software-based multimedia processing because they do not adapt well to time-triggered callback-based architectures. This paper has presented the *Time-Triggered SDF (TTSDF)*, a new model that mends those limitations because it: *a)* combines a set of actors associated to a (single) timed trigger with a set of actors which are untimed; *b)* retains token causality and dataflow semantics (arguably the added delay could be considered an exception), without compromising its calculability, which is equivalent to SDF; *c)* avoids jitter and does so by using an optimum amount of latency —a superior bound for that latency is also given by the model—; and finally *d)* naturally fits in callback-based architecture: the periodic dataflow scheduling is split into smaller sequences which can efficiently execute within the callback, avoiding external buffering and further threads. And offers further benefits such as: *a)* enables static analysis for optimum distribution of run-time load among callbacks; *b)* can be parallelized using well known techniques<sup>19</sup> also used in other dataflow models; and *c)* the time-triggered style of scheduling can be adapted to other dataflow models with quasi-static token rates, like *Boolean SDF*, or with dynamic (but explicit) token rates, like *Dynamic Dataflow*.

We have recently implemented this model in CLAM, a C++ open-source framework for audio processing<sup>20</sup> —previously, CLAM used a naive dynamic scheduler that only worked with simple multi-rate graphs. Several real-time multi-rate applications have been tested in the domains of audio features analysis, spectral audio transformation and acoustics simulation and 3D-audio generation.

We believe that multi-rate dataflows are not widely used in multimedia domains today (with exceptions, like Marsyas<sup>21</sup> and the CLAM framework) mainly because of the lack of useful abstractions and run-time frameworks. Leveraging the declarative style of dataflow with multi-rate capabilities promises more efficient systems (e.g. making good use of available multi-processors) and, not less important, more expressive and understandable abstractions. Previous efforts by the authors in this line includes a *dataflow pattern language*.<sup>22</sup> Some of the multimedia sub-domains that we believe might take advantage of multi-rate dataflows are: video coding and decoding, multi-frame-rate video processing, joint video and audio processing, audio processing in the spectral domain, real-time video and audio feature extraction using arbitrary sized token windows, and real-time computer graphics.

Many lines are open for future work. Open architectures that leverage parallelism and multiprocessors but are still compatible with callbacks for input and output should be studied. Such architectures would not see the users callback as a black-box function, but would have access to the dataflow graph declaration. Further, techniques for balancing the run-time load, not only between sequential callbacks, but multiple processors, should be developed.

## 7. ACKNOWLEDGEMENTS

This work is partly supported by the European Union (FP7 project 2020 3D Media ICT 2007), and a ICREA grant from the DIUE of the Catalan Government.

## REFERENCES

- [1] Lee, E. A. and Messerschmitt, D. G., “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Trans. Comput.* **36**(1), 24–35 (1987).
- [2] Lee, E. A. and Parks, T., “Dataflow Process Networks,” *Proceedings of the IEEE* **83**(5), 773–801 (1995).
- [3] Dennis, J., “First version of a data flow procedure language, Programming Symposium,” *Proceedings Colloque sur la Programmation*, 362–376 (1974).
- [4] Kahn, G. and MacQueen, D., “Coroutines and Networks of Parallel Processes,” *Information Processing 77, Proceedings of IFIP Congress* **77**(7), 993–998 (1977).
- [5] Geilen, M. and Basten, T., “Requirements on the execution of kahn process networks,” in [*Proceedings of the 12th European Symposium on Programming, ESOP*], (2003).
- [6] Edwards, S., Lavagno, L., Lee, E. A., and Sangiovanni-Vincentelli, A., “Design of Embedded Systems: Formal Models, Validation, and Synthesis,” *Readings in Hardware/Software Co-Design* (2001).
- [7] Bencina, R. and Burk, P., “PortAudio—an open source cross platform audio API,” *Proc. 2001 Intl. Computer Music Conf. (ICMC-01)* (2001).
- [8] Halbwachs, N., “Synchronous programming of reactive systems,” in [*Computer Aided Verification*], 1–16 (1998).
- [9] Ade, M., Lauwereins, R., and Peperstraete, J. A., “Data memory minimisation for synchronous data flow graphs emulated on dsp-fpga targets,” in [*DAC ’97: Proceedings of the 34th annual conference on Design automation*], 64–69, ACM Press, New York, NY, USA (1997).
- [10] Buck, J., *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, PhD thesis, University of California (1993).
- [11] Girault, A., Lee, B., and Lee, E. A., “Hierarchical finite state machines with multiple concurrency models,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* **18**(6), 742–760 (1999).
- [12] Kerihuel, A., Mcconnell, R., and Rajopadhye, S., “Vsdf: synchronous data flow for vlsi,” in [*Circuits and Systems, 1994., Proceedings of the 37th Midwest Symposium on*], **1**, 389–392vol.1 (3-5 Aug. 1994).
- [13] Pastrnak, M., Poplavko, P., de With, P., and Farin, D., “Data-flow timing models of dynamic multimedia applications for multiprocessor systems,” in [*System-on-Chip for Real-Time Applications, 2004. Proceedings. 4th IEEE International Workshop on*], 206–209 (19-21 July 2004).
- [14] Henzinger, T., Horowitz, B., and Kirsch, C., “Giotto: A Time-Triggered Language for Embedded Programming,” *Embedded Software: First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October 8-10, 2001: Proceedings* (2001).
- [15] Lee, E. A., “Modeling concurrent real-time processes using discrete events,” *Annals of Software Engineering* **7**(1), 25–45 (1999).
- [16] Wang, G. and Cook, P., “ChucK: a programming language for on-the-fly, real-time audio synthesis and multimedia,” *Proceedings of the 12th annual ACM international conference on Multimedia*, 812–815 (2004).
- [17] Lee, E. A. and Zhao, Y., “Reinventing Computing for Real Time,” *Proc. of the Monterey Workshop*, 1–25 (2006).
- [18] Eker, J., Janneck, J., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y., “Taming heterogeneity—the Ptolemy approach,” *Proceedings of the IEEE* **91**(1), 127–144 (2003).
- [19] Kohler, W., “A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems,” *IEEE Transactions on Computers* **24**(12), 1235–1238 (1975).
- [20] Amatriain, X., Arumi, P., and Garcia, D., “Clam: a framework for efficient and rapid development of cross-platform audio applications,” in [*MULTIMEDIA ’06: Proceedings of the 14th annual ACM international conference on Multimedia*], 951–954, ACM (2006).
- [21] Tzanetakis, G., [*Marsyas: a case study in implementing Music Information Retrieval Systems*], Idea Group Reference (2008).
- [22] Arumi, P., Garcia, D., and Amatriain, X., “A Data Flow Pattern Language for Audio and Music Computing,” *Proceedings of the 13th Pattern Languages of Programming* (2006).