Università degli studi di Udine

Ciaramella: A Synchronous Data Flow Programming Language For Audio DSP

Original

Availability:
This version is available http://hdl.handle.net/11390/1232165 since 2022-09-14T20:17:57Z

Publisher:

Published
DOI:10.5281/zenodo.6798221

Terms of use:
The institutional repository of the University of Udine (http://air.uniud.it) is provided by ARIC services. The aim is to enable open access to all the world.

Publisher copyright

(Article begins on next page)

17 November 2023

Ciaramella: A Synchronous Data Flow Programming Language For Audio DSP

Paolo Marrone
Orastron srl
paolo.marrone@orastron.com

Stefano D'Angelo
Orastron srl
stefano.dangelo@orastron.com

Federico Fontana
Università di Udine
federico.fontana@uniud.it

Gennaro Costagliola
Università di Salerno
gencos@unisa.it

Gabriele Puppis
Univesità di Udine
gabriele.puppis@uniud.it

ABSTRACT

Various programming languages have been developed specifically for audio DSP in the last decades, yet only a handful of industrial and commercial applications are known to actually use them. We assume that this is due to some common deficiencies of such languages, namely the tight coupling between syntax and computational model, which limits modularity, and the adoption of programming paradigms that are conceptually distant from conventional DSP formalism. We propose a new audio DSP programming language, called Ciaramella, based on the synchronous data flow (SDF) computational model and featuring a fully declarative syntax to address these issues. A source-to-source compiler which translates Ciaramella code to C++ and MATLAB programs has been developed. We have checked that our solution allows to naturally represent and correctly schedule highly-interdependent DSP systems such as Wave Digital Filters (WDFs) which would be hard to handle in current audio DSP languages.

1. INTRODUCTION

Audio programming languages raise interest in both the academia and the industry. They facilitate DSP systems programming by providing high-level abstractions to, e.g., efficiently manipulate data streams. There exist numerous audio programming languages and environments whose development dates back to the sixties, and they offer heterogeneous syntax, programming and computational models, and compilers or interpreters.

If the target is to develop computationally efficient audio code, then relevant and actively developed languages are FAUST [1], Max/Gen [2], Reaktor core [3], Soul [4] and Kronos [5]. FAUST is a mature language which adopts a purely functional paradigm, whose compiler performs extensive optimizations and translates to a wide set of plugin APIs. Gen from the environment Max is a visual programming language which combines graphical representation with textual instructions in a declarative fashion. Gen patches can be translated into VST plugins. Reaktor core, from the Reaktor music software studio, allows visual programming of DSP systems similarly to Max Gen but is not suitable for building plugins. Soul, a relatively recent project, provides a set of features for coding and running audio DSP systems; in particular, its language intentionally adopts a C++-like style improved with some instructions for working with data flows. Kronos, defined as a *metalanguage*, explores the declarative approach to facilitate the manipulation of signals, graphic visualization and just-in-time compiling. Conversely, SuperCollider and ChucK put emphasis on providing an immediate sound result, by featuring code interpretation and live coding.

Despite the existence of such domain specific languages, the music software industry tends to rely on general-purpose programming languages like C and C++ [6], although they force developers to pay attention to a number of details such as memory management, implementation techniques, compliance with plugin standards, and optimizations, which could rather be efficiently handled by domain-specific tools automatically. In our opinion, a potentially successful language should exclusively focus on the description of audio DSP systems using a limited set of domain-specific abstractions that are familiar to users and should be as modular and flexible as possible. At the same time, the compiler of such language should produce code that is easily embeddable in products that run on a wide range of platforms.

Unfortunately, none of the languages mentioned above meets all such requirements. For example, besides their differences, they share a syntactic rigidity by which it is impossible to describe delay-free feedbacks between components, whether built-in or composite (e.g., the subpatches of Gen): either a syntactical error is thrown or the feedback operation automatically implies the addition of a unit delay. In the former case, not only it is impossible to describe uncomputable systems containing delay-free loops, but also those computable ones containing loops in which delays are "hidden" inside a composite component. In the latter case, the implicit addition of a delay at a compositional level tightly couples the internal implementation details of each component to its external usage, which clearly violates encapsulation principles. Both arrangements prevent from easy de-

Copyright: © 2022 Paolo Marrone et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

scription and modular composition of multi-directional structures such those found in WDFs [7].

These practices are documented in the recursive composition section in the FAUST manual [1], in section 4.10 of the guide to Reaktor 6 core [2], in the feedback loops section of the SOUL documentation [3], in the History command section of Max/Gen [4] and in [5] for Kronos.

The proposed language addresses this problem at a semantic rather than syntactic level. It provides an unconstrained syntax based on a fully declarative approach and a dataflow block-based representation which should be familiar to DSP programmers. The compiler backend first *flattens* described systems into a representation consisting only of built-in components and interconnections, and then only at such a late stage evaluates and ensures their computability. Also, the underlying semantics of Ciaramella leverage the Synchronous Data Flow (SDF) framework which has been extensively researched [8] but not yet directly or fully adopted by current audio languages, despite allowing to naturally represent data flows and the mathematical operations on them. Finally, the compiler produces generic C++ and MATLAB code with no external dependencies. Ciaramella, however, is still at an experimental stage and it is far from being feature complete.

The paper is organized as follows. Section 2 recalls some theoretical facts about Synchronous Data Flow, reporting the current state of art of the SDF programming languages and showing how the SDF model fits the DSP domain. Section 3 describes the language concepts, abstractions, and syntax, furthermore showing some basic examples. Section 4 describes the compiler developed for Ciaramella, by focusing on the internal representation of the process topology, optimizations, computability verification, scheduling, and output code generation phases. Section 5 shows the implementation in Ciaramella of a simple WDF low pass filter.

2. SYNCHRONOUS DATA FLOW MODEL

SDF [8] is a restriction of Kahn Process Networks (KPN) [9]. It provides a set of primitives for describing a network of independent processes (also called *actors*) that communicate with each other via unidirectional FIFO queues emitting and consuming data values, called *tokens*. An *execution* is ruled by few constraints:

- writing (on a queue) is non-blocking;
- reading (from a queue) is blocking;
- reading implies token consumption;
- a queue can be written by only one process;
- a queue can be read by only one process;
- the number of tokens read and written by each process per queue per execution is known in advance.

Interestingly, the queues can be initialized with some tokens before the first execution. This corresponds to introduce a communication delay between two actors. An SDF *state* is defined as the number of tokens stored in every queue of the network.

The last rule differentiates SDF from KPN, and it is of crucial importance since it permits static scheduling at compile time [10] of the network for a correct sequential, or even parallel, execution. An SDF schedule is defined as a sequence of actor firings repeated periodically, and each repetition is called *period*. A periodic schedule is a finite schedule that invokes each actor at least once and does not change the SDF state: this guarantees that the tokens are not accumulated in the queues. Finally, a valid schedule is a periodic schedule which does not cause a graph deadlock, i.e. an actor planned for execution is not fireable if the input is unavailable. If, for a SDF graph, there is at least one valid schedule, the graph is defined to be *consistent*.

The SDF model is inherently multi-rate since the actors can produce and read different numbers of tokens at each firing. An actor generally performs simple operations like sum, multiplication, division, where it consumes two input tokens and produces one output token. However, *composite* actors have been defined [11] for modularity and compositional modelling, too. They encapsulate an SDF graph and act like normal actors, featuring readability and allowing to handle more complex systems easily. Conversely, non-composite actors are said to be *atomic* and a network composed only by atomic actors is called *flat*, so that the process of substituting the composite actors with their internal graph is said *flattening*.

2.1 SDF Languages

The SDF model inspired the development of some related programming languages in the last decades. They are thought mainly for synchronous reactive systems, which continuously interact with the execution environment at the speed imposed by it. The most relevant are Lustre [12], SIGNAL [13] and Esterel [14], all born in the eighties. Lustre gained some commercial success for some critical systems, too [15]. They all allow for the description of an SDF network in terms of actors and connections; most importantly, they provide temporal operators for accessing past values (delay) and for setting the initial value of the communication queues. Barkati *et al.* [16] made an especially important work for our study, since they analysed 10 programming languages, 5 of which are audio-specific and 5 SDF-specific, including the aforementioned ones. By comparing their syntax and expressiveness while implementing a digital oscillator, they showed, e.g., how

[1] https://faustcloud.grame.fr/doc/manual/index.html#recursive-composition
[2] https://www.native-instruments.com/fileadmin/ni_media/downloads/\manuals/REAKTOR_6_Building_in_Core_English_2015_11.pdf
[3] https://github.com/soul-lang/SOUL/blob/master/docs/SOUL_Language.md#feedback-loops
[4] https://docs.cycling74.com/max6/dynamic/c74_docs.html#gen_overview

correspondingly different notions of time were conceptualized: in SDF languages time is in fact more strictly an abstract logical notion, using multiple clocks to perform operations at different rates; on the other hand, audio languages adopt a less general and more practical approach directly using the sampling rate information. Concerning programming paradigms, while Esterel is imperative, Lustre and SIGNAL rather go for the declarative approach, in particular the equational and the relational ones, respectively. The SDF languages are descriptive à la VERILOG, leaving the scheduling of the processes to the compiler [10]. In summary, the study shows a close correlation between the audio and the synchronous domains by giving an overview of how they represent a typical DSP problem. We further investigate this relation by designing an audio programming language that fully adopts the SDF model.

For a detailed survey about synchronous programming of reactive systems check [17].

3. THE CIARAMELLA LANGUAGE

Ciaramella is an audio domain-specific programming language that adopts the SDF model for describing the DSP part of an audio plugin. Its design goals are:

- simple and unconstrained syntax;
- modularity;
- composability.

In order to meet them we chose the declarative paradigm, as opposed to the imperative one: this way, the order of the instructions is not relevant and a "variable" cannot be assigned more than once according to the static single assignment (SSA) form [18]. This makes the assignment operator semantically comparable to equivalence. Also, any expression in Ciaramella including identifiers always refers to streams/flows of data: for instance, the statement $a = b + c$ means $a_n = b_n + c_n$ at temporal sample n.

Our aim regarding the semantics of Ciaramella is to provide a minimal set of programming abstractions that is sufficient to represent SDF systems, as described in the following subsections.

3.1 Blocks, Ports, Connections

In Ciaramella there are three main components:

- Block: it represents an SDF actor and it encapsulates an operation. A list of input ports and a list of output ports are attached to it. Every block reads tokens from the input ports and writes tokens to the output ports.
- Port: it is a communication endpoint. It can be of input or output type.
- Connection: it defines a directed flow of data (tokens) between two ports. Together with the port, it is analogous to the SDF queue.

An output port can be shared by multiple connections. The current implementation of Ciaramella provides the following kinds of block.

3.1.1 Elementary Operation Blocks

Sum, subtraction, multiplication, division, sign change. They have 2 input ports and 1 output port except for the sign change/unitary minus which has 1 input port.

3.1.2 Variable, Numeric and Sample Rate Blocks

A variable block has 1 input and 1 output port and only works as a fork for data flows. A numeric block has only 1 output port and acts like a constant source of a given numeric value. The sample rate block is a special numeric block which outputs the sampling rate as dictated by the execution environment. It is accessible by the `fs` keyword at any point of the code.

3.1.3 Unit Delay Block

A unit delay has 1 input and 1 output port. Its operation is to store the nth input token and to write the $(n-1)$th one. In a pure SDF system a unit delay is obtained by pre-initializing a queue with one token. Operations on the connections in the SDF formalism are replaced by the unit delay block operation in Ciaramella in order to keep the language simple and the program structure clean. The two approaches are equivalent.

3.1.4 Composite Block

Composite blocks define sub-systems containing other blocks and connections and exposing a variable number of input and output ports. They are analogous to Nodes in Lustre [12] and subpatches in Max [2]. In a Ciaramella program there must always be at least one composite block which acts like a "main", and it is the only one that directly interacts with the execution environment.

3.2 Syntax

Ciaramella also aims at syntactic minimalism and essence by having a low number of keywords and adopting a conventional and simple design of the program structure. A Ciaramella program is made of a list of constant assignments with global scope and a list of composite block definitions with an internal local scope. The body of a composite block definition consists of a list of assignments. The following subsections provide more details.

3.2.1 Assignments, Expressions, Types

Assignments and expressions follow a C-like syntax. There are no explicit types, rather all values are meant to be in IEEE 754 floating point representation [19]. A composite block is defined by a list of output ports, its identifier, the list of input ports, and the body. The following example defines a stereo volume controller composite block:

```
y1, y2 = stereoVolCtrl(x1, v1, x2, v2){
    A = 0.8
    y1 = x1 * v1 * A
    y2 = x2 * v2 * A
}
```

`y1` and `y2` are the output ports; `stereoVolCtrl` is the name of the composite block; `x1`, `v1`, `x2` and `v2` are the input ports; the body is included between { and }. Within the body of the composite block all the declared output ports must be assigned and they can be used as variables within expressions. The input ports cannot be assigned since their value is set outside the block.

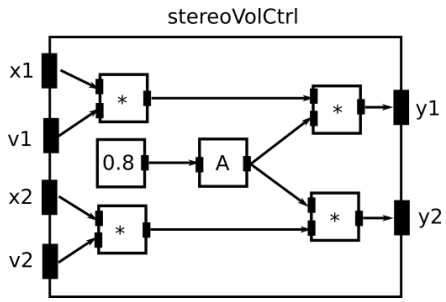

Figure 1: A representation of the `steroVolCtrl` composite block example.

Figure 1 contains a graphical representation of the `stereoVolCtrl` composite block. It shows how every expression creates a block, and sub-expressions are automatically connected to their parent expression; also, assignments determine connections between expressions and assigned variables.

3.2.2 Modularity and Composition

Defining composite blocks allows for reuse and modularization of the code. A composite block can be *instantiated* within another using a syntax resembling a function call in C. In the following example,

```
y = VolAttenuator(x1, x2) {
    t1, t2 = stereoVolCtrl(x1, 0.1, x2, 0.2)
    y = t1 + t2
}
```

the composite block `VolAttenuator` instantiates a `steroVolCtrl`. Intuitively, the first output port of `steroVolCtrl` gets connected to the input port of `t1` and the second one to `t2`, as shown in figure 2.

3.2.3 Unit Delays

Unit delays are the analogue of the *pre* operator in Lustre. It is fundamental to create explicitly computable loops within the SDF network. The expression `delay1(x)` returns x_{n-1}, i.e. the previous value. For example,

```
counter = delay1(counter) + 1
@counter = 0
```

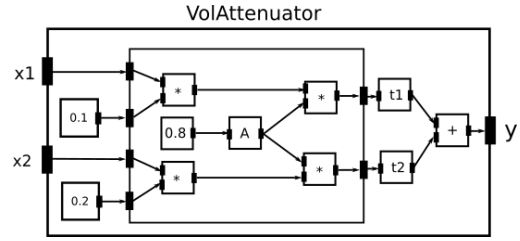

Figure 2: A representation of the `volAttenuator` composite block example.

implements a counter and it is equivalent to the recursive formula $counter_0 = 0$; $counter_n = counter_{n-1} + 1$ for $n \geq 1$. The second statement sets the initial value of `counter`.

By nesting unit delays, one can define a multiple delay as in

```
fib = fib_pre + delay1(fib_pre)
fib_pre = delay1(fib)
@fib_pre = 1
@fib = 0
```

which calculates the Fibonacci number and corresponds to

$$\begin{aligned} fib_{-1} &= 1 \\ fib_0 &= 0 \\ fib_n &= fib_{n-1} + fib_{n-2}. \end{aligned}$$

Note that these two examples are purely demonstrative are actually useless in audio applications.

3.3 Initialization Statement

We have shown how the `@A = expr` syntax sets the initial value of a variable. In particular, `expr` can be any valid Ciaramella expression, yet in this context the values of each variable in `expr` is evaluated as follows: if it is itself assigned an initial value expression, then this algorithm is applied recursively; otherwise, if there is a regular assignment expression in the current block, that is recursively used; in all other cases, the variable is necessarily external, hence its value is externally determined (it evaluates to 0 if the variable is a "main" block input). All delay operators are ignored.

In the following example

```
@A = 1 + B * C - x
@B = 3
B = x * 0.5
C = 2 + 5
```

`x` is an external audio rate signal, hence the initial expression of `A` evaluates to `@A = 1 + 3 * (2 + 5) - 0`.

3.3.1 Comments, End Statement

In Ciaramella a statement can be ended by either a semicolon or a new line. There are two kinds of comment: a classic single-line comment starting with a "#" and a

continuation comment starting with three dots, "`...`", lasting until the end of line, and which lets the statement continue on the next line.

4. ZAMPOGNA: THE CIARAMELLA COMPILER

We developed a small yet functional compiler for Ciaramella called Zampogna, sized ∼1000 lines of JavaScript code. The compilation process consists of several subsequent steps: the first ones are parsing and extended syntax validation, whose output is an Abstract Syntax Tree (AST). The AST is used to generate an intermediate representation graph (IG), corresponding to the SDF process network graph. Upon this graph, optimizations and static scheduling are performed. Finally, the compiler produces target code from the schedule.

4.1 AST to IG

The IG exactly corresponds to the set of components mentioned in Section 3.1. The compiler translates the AST to the IG by converting expressions to blocks and connections, recursively inserting sub-system instances while appropriately connecting them, and finally by flattening the resulting graph as described in Section 2. Figure 3 shows the flattened IG for the previous `VolAttenuator` example. Recursive block instantiation is not allowed.

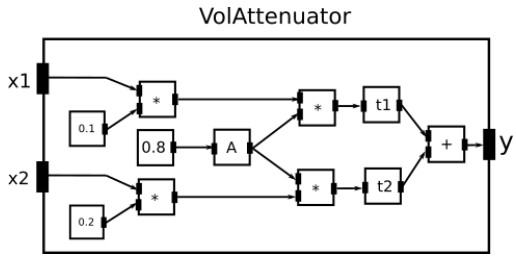

Figure 3: A representation of the `VolAttenuator` composite block example after the flattening operation.

The initialization statements (`@id = expr`) are not involved in the construction of the IG but, conveniently, the compiler uses them to produce another intermediate graph, called *Intermediate Initialization Graph* (IIG) which is executed only once, prior to the main IG, to produce initial values.

4.2 Optimizations

The compiler performs a series of optimizations over the IG. Some of them are common like dead code elimination, constant propagation, and copy propagation. Domain-specific optimizations are more interesting, particularly those based on value update rates. A typical audio application, besides processing audio signals, receives asynchronous input events at runtime. Usually the time distance between two of such events is significantly higher than the sampling period of the audio signals. In the SDF model, asynchronous events are treated as normal synchronous signals whose value is externally set. This keeps the domain coherent while providing to the programmer a unified view of signals, events, and constants. Indeed, every block of the graph emits one sample at each firing, regardless of its nature. For example, numeric constant blocks output the same token every time, audio rate blocks produce (potentially) different tokens at each firing and user control input blocks emit the same token until the user moves the associated knob or slider. Being aware of such characteristic of the blocks, we define the *update class* property which is necessary for the compiler to operate optimizations on the final code.

Four update classes are defined, ordered by increasing update frequency:

- **Constants class**. Numeric constants belong to this class.
- **Sample rate class**. The sampling rate value is set by the execution environment at runtime. This is typically done only once. `fs` block output falls into this update class.
- **Control rate class**. The output of the blocks representing user input controls are included in this class.
- **Audio rate class**. Input and output audio signals are part of this class, as well as the output of the `delay1` blocks.

The update class information is attached to the ports rather than the blocks since it is, conveniently, a property of the signals. It propagates to all nodes of the network starting from constants, `fs`, inputs and `delay1`. Recursively, the output port of a block inherits the highest rate of the input ports it depends on. Conversely, input ports inherit such property from the ports they are connected to.

Leveraging this information, the compiler makes sure that intermediate and output values are only updated when a change occurs or might occur. This behaviour is commonly known in other contexts as lazy evaluation of expressions.

4.3 IG Static Scheduling

After the optimizations over the IG are performed, the next phase is the scheduling of the atomic blocks for sequential execution. Such blocks need to be scheduled taking into account their mutual dependencies. In particular, it is possible to build a dependency graph (DG) starting from the IG: every directed arc of the IG, except for those starting from `delay1` blocks, define a dependency. If $A \rightarrow B$ indicates a connection from an output port of A to an input port of B, and if A is not a `delay1` block, then A must be executed prior to B; we say that B *has an instantaneous dependency on* A. Delay blocks are excluded because their output value is already known at the beginning of the n-th (current) iteration, as it was calculated at the $n-1$-th (previous) one. There can be multiple valid schedules and [10] investigates the

issue for multi-rate SDF graphs. Our case is simpler since we manage only a single synchronous rate and all blocks produce and consume only one token at each firing. Therefore, a valid schedule can be represented by a stack which can be filled by recursively removing the nodes that do not have instantaneous dependencies left and pushing them onto the stack.

If an instantaneous circular dependency path is encountered, the compiler throws an error. This arrangement is commonly known in the audio DSP literature as *delay-free loop* and is not computable. Unlike other languages, the ability to verify this condition at such a late stage, that is after having fully flattened the IG into atomic operations and having attempted scheduling on the full IG, frees us from all syntactical and semantic restrictions in representing instantaneous feedbacks. This allows for natural description of highly-interdependent DSP systems such as WDFs.

As an example, we implemented a wave digital filter in section 5: its unflattened graph appears to be uncomputable due to the presence of delay-free loops, while, when flattened, it is perfectly computable because delays are found within the composite blocks.

4.4 IIG Static Scheduling

The IIG is scheduled similarly to the main IG, but with an important difference: the `delay1` blocks are not excluded from the construction of the arcs of the DG as no previous value can be used. Therefore, in the IIG there cannot exist loops, no matter whether they contain delays or not.

4.5 Output Code Generation

Zampogna produces a C++ program with a VST2 wrapper, and a MATLAB script which is useful for fast prototyping. We used the doT JavaScript templating library [20] to accomplish this, which makes the code generation part of the compiler modular and easily extensible.

4.6 Compiler Options

In order to keep the syntax simple and the programs modular, some information can only be passed to the compiler via command line options. For example, the name of the "main" block and the list of input ports associated to user controls belong to this set of data.

```
Usage: zampogna.js [-i initial_block]
    [-c control_inputs] [-v initial_values]
    [-t target_lang] [-o output_folder]
    [-d debug_bool]
    input_file
```

4.7 Implementation

The implementation of the Zampogna compiler is available at https://github.com/paolomarrone/Zampogna. It has been developed for NodeJS and the only external dependencies are doT [20] for the output code generation and Jison [21] for the parser code. It comes with a few examples, like the WDF from Section 5.

5. A WAVE DIGITAL FILTER IMPLEMENTATION

As a case study we implemented a low pass transfer function using a WDF [22]:

```
pi = 3.141592653589793
b, R0 = wdf_resistor(a, R) {
    b = 0
    R0 = R
}
b, R0 = wdf_capacitor(a, C) {
    b = delay1(a)
    R0 = 0.5 / (C * fs)
}
b = wdf_voltage_source_root(a, E) {
    b = 2 * E - a
}
bu, bl, br, R0 = wdf_3port_series(au, al, ar, Rl, Rr) {
    bl = al - Rl / (Rl + Rr) * (al + ar + au)
    br = ar - Rr / (Rl + Rr) * (al + ar + au)
    bu = -(al + ar)
    R0 = Rl + Rr
}
y = lp_filter(x, cutoff) {
    fc = (0.1 + 0.3 * cutoff) * fs
    C = 1e-6
    R = 1 / (2 * pi * fc * C)

    bR, RR = wdf_resistor(aR, R)
    bC, RC = wdf_capacitor(aC, C)
    bV = wdf_voltage_source_root(aV, x)
    aV, aR, aC, Rp = wdf_3port_series(...
        bV, bR, bC, RR, RC)
    @aC = 0

    y = 0.5 * (aC + bC)
}
```

which can be compiled by the following command

```
zampogna.js -i lp_filter -c cutoff
    -t cpp lp_filter.crm
```

The composite block `lp_filter` instantiates other composite blocks, which mutually exchange signals by recursively creating loops. The compiler does not add implicit delays: the only unit delay is specified in `wdf_capacitor`. Even if at a first glance it may seem that the WDF gives rise to delay-free loop errors, indeed the system is found to be explicitly computable thanks to the flattening operation over the IG and the global scheduling. Figure 4 shows a simplified graphical view of the unflattened IG of `lp_filter`.

This case study shows how Ciaramella is able to natively represent a class of DSP problems, such as WDFs, that is known to be cumbersome to program in any other audio programming language. Furthermore, it demon-

strates its high modularity and composability characteristics.

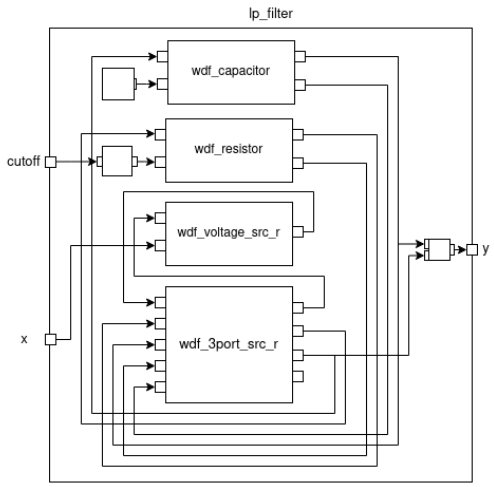

Figure 4: Simplified and unflattened graphical representation of the `lp_filter` WDF example.

Support for the implementation of wave digital models has been recently added to FAUST through the *WD-models* library [7]. It is based on an ad-hoc language extension referred as metaprogramming which uses a declarative style. While it manages to represent WDF systems and produce efficient code, we argue that it lacks modularity and flexibility due to the automatic inclusion of unit delays by the feedback operator ($\sim$). All necessary component-level delays are implemented at the tree level, in the most simple case by the `buildtree(connection_tree)` function, whose code reads [5]

```
builddown(connection_tree)~buildup(connection_tree) :
buildout(connection_tree)
```

In order for this to work properly, delays must be not implemented in the leaf component they would naturally belong to. For example, the capacitor component is implemented by the authors as

```
capacitor =
case{
        (0, R) => _*1;
        (1, R) => _;
        (2, R) => R0
        with {
                R0 = t/(2*R);
        };
}with{
        t = 1/ma.SR; //sampling interval
};
```

where clearly there is no delay between input and output, contrary to the standard WDF formulation.

[5] `https://github.com/grame-cncm/faustlibraries/blob/master/wdmodels.lib`

6. CONCLUSIONS

We presented the syntax and semantics of Ciaramella, a novel audio DSP programming language, and gave an overview of its compiler. Ciaramella combines a simple and declarative syntax with the SDF computational model to represent audio DSP systems. This choice results in an high level of modularity and composability which makes it possible to natively represent even complex systems such as WDFs.

The Ciaramella language and its compiler are however still at an early stage of development. Further work could include the implementation of conditional and loop blocks, multi-rate support, a library of common mathematical functions, n-delay operations, arrays and matrices, as well as more optimizations by symbolic simplification of expressions, pattern recognition techniques, etc.

7. REFERENCES

[1] "FAUST," https://faust.grame.fr/, accessed: 2022-02-08.

[2] "Max/gen," https://docs.cycling74.com/max7/vignettes/gen_topic, accessed: 2022-02-08.

[3] "Reaktor core," https://www.native-instruments.com/fileadmin/ni_media/downloads/manuals/REAKTOR_6_Building_in_Core_English_2015_11.pdf, accessed: 2022-02-08.

[4] "Soul," https://soul.dev/, accessed: 2022-02-08.

[5] V. Norilo, "Kronos: a declarative metaprogramming language for digital signal processing," *Computer Music Journal*, vol. 39, no. 4, pp. 30–48, 2015.

[6] S. D'Angelo, "Lightweight virtual analog modeling," in *Proceedings of the 22nd Colloquium on Music Informatics, Udine, Italy*, 2018, pp. 20–23.

[7] D. Roosenburg, E. Stine, R. Michon, and J. Chowdhury, "A wave digital filter modeling library for the FAUST programming language," 6 2021.

[8] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[9] K. Gilles, "The semantics of a simple language for parallel programming," vol. 74, 1974, pp. 471–475.

[10] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 24–35, 1987.

[11] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee, "Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, pp. 1–26, 2013.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[13] T. Gautier, P. L. Guernic, and L. Besnard, "Signal: A declarative language for synchronous programming of real-time systems," in *Conference on Functional Programming Languages and Computer Architecture*. Springer, 1987, pp. 257–277.

[14] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[15] N. Halbwachs, "A synchronous language at work: the story of lustre," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05*. IEEE, 2005, pp. 3–11.

[16] K. Barkati and P. Jouvelot, "Synchronous programming in audio processing: A lookup table oscillator case study," *ACM Computing Surveys (CSUR)*, vol. 46, no. 2, pp. 1–35, 2013.

[17] N. Halbwachs, "Synchronous programming of reactive systems," in *International Conference on Computer Aided Verification*. Springer, 1998, pp. 1–16.

[18] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988, pp. 12–27.

[19] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[20] "dot - the fastest + concise javascript template engine for node.js and browsers." https://olado.github.io/, accessed: 2022-02-07.

[21] "Jison," https://github.com/zaach/jison, accessed: 2022-02-07.

[22] A. Fettweis, "Wave digital filters: Theory and practice," *Proceedings of the IEEE*, vol. 74, no. 2, pp. 270–327, 1986.