Introducing stateful conditional branching in Ciaramella

Paolo Marrone
Orastron s.r.l
Università degli Studi di Udine
paolo.marrone@orastron.com

Stefano D'Angelo
Orastron s.r.l.
stefano.dangelo@orastron.com

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

ABSTRACT

Conditional branching in Synchronous Data Flow (SDF) networks is a long-standing issue as it clashes with the underlying synchronicity model. For this reason, conditional update of state variables is rarely implemented in data flow programming environments, unlike simpler selection operators that do not execute code conditionally. We propose an extension to SDF theory to represent stateful conditional branching. We prove the effectiveness of such approach by adding conditional constructs to the Ciaramella programming language without compromising its modular declarative paradigm and maintaining domain-specific optimizations intact. This addition enables easy implementation of common DSP algorithms and helps in writing efficient complex programs.

1. INTRODUCTION

Ciaramella [1] is an high-level programming language for coding audio DSP systems. Its declarative syntax and semantics are fully compliant with the Homogeneous Synchronous Data Flow (HSDF) computational model [2], yet with a special emphasis on modularity and flexibility. This combination allows for straightforward coding of even complex DSP systems.

A source-to-source compiler called Zampogna [1] was developed in JavaScript, which parses Ciaramella code. Zampogna creates an internal graph (IG) representation of the input DSP system as a Synchronous Data Flow (SDF) model; then, it flattens, optimizes, and statically schedules [3] the IG; finally, it produces the corresponding C++, MATLAB, D, or JavaScript program. In doing so, Zampogna statically schedules the SDF system, that is, it finds a fixed computable execution (firing) order of the dataflow network nodes. This implies determinism and, as a practical consequence, efficiency of the resulting software application, which may not be true of systems relying on dynamic scheduling [4].

Conditional branching is a long-standing issue related to control flow in SDF models, having direct practical consequences in audio programming. In essence, every node in an SDF process network must always be fired so as not to break the synchrony by unbalancing the production/consumption of tokens. Token balancing prevents from selective execution of nodes, with the result that all branches must be always executed. Operations whose outcome depends on a condition are usually achieved by introducing a selection operator, which takes all possible outcomes as input and forwards the "selected" one to output based on the input condition, in analogy with electronic multiplexers. In such a scenario, not only all branches are always executed, causing potentially significant performance penalties, but also states in all branches are always updated, which makes it cumbersome to sensibly implement common algorithms that rely on conditionally-updated states or branches, such as decimators and polyphase filters.

A few extensions to SDF theory have been proposed to tackle this issue, e.g., by allowing scheduling of mixed systems using quasi-static scheduling [5], yet increasing complexity and at the expense of modularity. Existing SDF programming languages, such as Lustre [6] and Signal [7], do not offer actual branching constructs but are limited to simple *select* instructions. Similarly, current audio-specific programming languages typically offer constructs with similar expressive power. FAUST [8] provides *selector* primitives [2] while new approaches to selective execution are being currently investigated [3]. Max [4] and its Gen extension also limit to offering a number of select-like operators [5]. Reaktore Core [6] supplies users with the *Router* module for routing signals and controls towards different processor blocks. In this case it is possible to conditionally execute whole branches, yet the internal working of the software is not publicly known. For example, it is hard to tell if blocks are scheduled statically or dynamically.

In this paper we propose a novel theoretical framework to transparently handle conditionals within the HSDF model while preserving modularity. We also extend the syntax of Ciaramella to represent *if-then-else* constructs. This syntax underpins the new theoretical solution within the HSDF formalism. We analyse some peculiar cases and how the compiler handles them.

[1] https://github.com/paolomarrone/zampogna

[2] https://faustdoc.grame.fr/manual/syntax/#selector-primitives

[3] https://github.com/orlarey/faust-ondemand-spec

[4] https://docs.cycling74.com/max8/

[5] https://docs.cycling74.com/max8/vignettes/gen_topic

[6] https://www.native-instruments.com/fileadmin/ni_media/downloads/manuals/REAKTOR_6_Building_in_Core_English_2015_11.pdf

The paper is organized as follows. Section 2 reminds basic concepts of SDF, recalls existing theory, and describes our theoretical solution regarding conditionals. Section 3 illustrates the new syntax additions to Ciaramella. Section 4 describes how dynamic scheduling is avoided in certain corner cases. Section 5 concludes the paper and suggests possible future developments.

2. SDF-COMPATIBLE CONDITIONAL BRANCHING

HSDF is a restriction of SDF, which in turn is a restriction of the Kahn Process Network (KPN) distributed computation model. A KPN graph is made of a set of independent and deterministic sequential processes (nodes) and a set of unidirectional FIFO communication queues (edges). A process can write/produce and/or read/consume tokens (samples) to/from the queues. KPN requires the following restrictions:

- writing on a queue is non-blocking;
- reading from a queue is blocking and implies token consumption;
- a queue can be written by only one process;
- a queue can be read by only one process.

SDF requires one more rule:

- the number n of tokens that are read and written by each process per queue and per execution is known in advance.

In HSDF, n must be equal to 1.

SDF and, consequently, HSDF models are appealing because they allow for static scheduling [3]. Lee and Messerschmitt proved that an HSDF graph can always be created by replicating the nodes of a general SDF graph [2]. This means that HSDF is not a restriction of SDF, since what is true for HSDF holds also for SDF. Ciaramella adopts HSDF in order to keep simplicity without loss of generality.

Conditional execution of nodes in SDF breaks the rule of emitting/reading a constant number of tokens. Figure 1 shows how a typical *if-then-else* construct could be built using two asynchronous nodes, `switch` and `select` [2]. The pair `(0,1)` means that the number of tokens read or written from/to adjacent queues can be 0 or 1, which breaks synchrony.

Otherwise, nodes `switch`, `f(·)`, `g(·)`, and `select` could be wrapped in a single node to be treated as a *black box*. This solution, however, breaks modularity in case a delay is hidden within the black box and a loop from its output to its input is established externally. Without knowledge of the internals of the black box, it would be impossible to establish whether a computable path exists.

Another possibility consists in executing all branches and select the desired output via the final `select` while discarding all others. In our example this would cause no inconsistencies and conditional execution may be fully restored, in theory, at a later code optimization stage. However, that would not be the case if a branch contained some state (e.g., due to a delay operation) which would need to be either updated or not depending on the *if-then-else* condition.

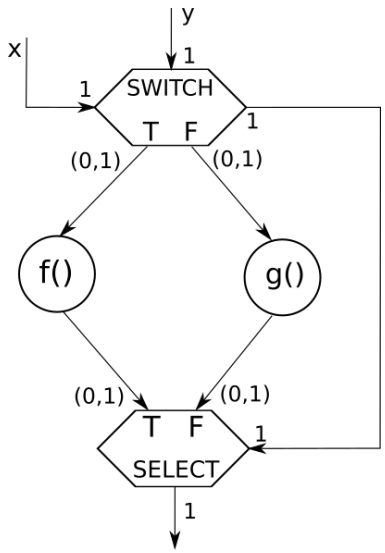

Figure 1: Conditional model proposed in [2]. `switch` and `select` are asynchronous nodes implementing the classic *if-then-else* statement. `(0,1)` means that nodes can read or write 0 or 1 tokens per execution. This model is, therefore, not synchronous.

In order to overcome such difficulties, we introduce a new node type called `conditional delay`.

2.1 Conditional delay

In SDF, a unitary delay can be obtained by initializing a queue with a token prior to the first graph execution [2]. After each graph execution that queue will always contain one token. This is the only way an SDF system can maintain state between executions.

In [1] we simplified the expression of a delay by avoiding queue pre-initialization and introducing a new node type called `delay1 block`, see Figure 2b. A `delay1 block` reads from the input queue, i_1, writes to the output queue, o_1, and contains the state s. Such operations are executed in the following order:

1 write s to o_1;
2 read t from i_1;
3 $s \leftarrow t$;

in which `s` was initialized before the first execution.

We now introduce a new node type called `conditional delay1 block`. While `delay1 block` has one input and one output, `conditional delay1 block` reads from $n+1$ input queues, $i_1, i_2, ..., i_{n+1}$, writes to one output queue, o_1, and contains a state s. Queues $i_2, ...i_{n+1}$ carry boolean tokens. It executes the following operations:

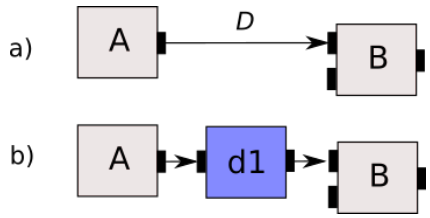

Figure 2: a) classic representation of a delay over a queue. b) analogous delay in `delay1 block` form. `A` and `B` are generic SDF process nodes.

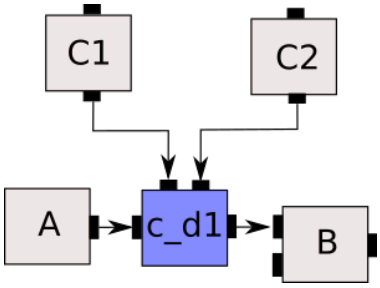

Figure 3: `conditional delay1 block`. `A` and `B` are generic processor nodes; `C1` and `C2` write boolean tokens on their output queues.

1 read $b_1, ..., b_n$ from $i_2, ...i_{n+1}$;
2 write s to o_1;
3 read t from i_1;
4 **if** $\bigwedge_{j=1}^{n} b_j$ **then**
5 $\quad s \leftarrow t$;
6 **else**
7 $\quad$ discard t;
8 **end**

Therefore, a `conditional delay1 block` behaves like `delay1 block` if the condition inputs are *true*, otherwise it does not update its state s. In fact, it can be seen as a generalization of `delay1 block`, to which it is equivalent when $n = 0$.

2.2 if-then-else based on conditional delays

Our solution for stateful conditional branching is based on the `conditional delay1 block`. To this aim, we introduce the `decoder` and `select` blocks. `decoder` reads tokens from an input queue and evaluates them as boolean conditions. It sends boolean `true` or `false` to the `conditional delay1 block`s that are part of the corresponding branches. In the example in Figure 4, if the output of `C` evaluates to `true` then `decoder` sends `true` to the `c_d1` block on the left (true) branch, and `false` to the block on the right (false) branch. Furthermore, `select` receives the output of the condition evaluation in order to forward the output from the chosen branch and discard those from other branches. A `conditional delay1 block` can depend on multiple conditional inputs in case of nested *if-then-else* constructs.

Under this scheme, all nodes in the process network are always executed, no matter which branch they belong to. Only the states in the selected branches will be updated

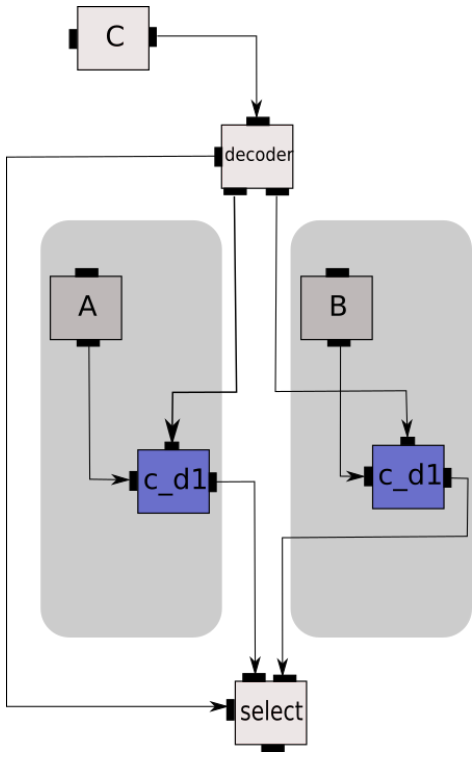

Figure 4: Conditional branching using `conditional delay1 block`, `switch` and `select`. The blocks on the light grey background belong to conditional branches

and only the proper branch outputs will be forwarded by `select` nodes. The process network can thus be statically scheduleded as usual [1, 3]. In actual implementations, it is also possible to avoid executing entire branches in the final output code by generating conditional constructs every time a `select` is encountered during the scheduling phase, while keeping track of dependencies.

3. EXTENDING CIARAMELLA SYNTAX AND SEMANTICS

Ciaramella adopts a fully declarative syntax. A program reflects the typical semantics of a HSDF graph. The programming style is also block-oriented. A simple example is the following implementation of a three poles low pass filter:

```
1  b = 0.1
2  y = lp (x) {
3      y_z1 = delay1(y)
4      y = y_z1 + b * (x - y_z1)
5      @y = 0
6  }
7
8  yL, yR = lp3 (x) {
9      yL = lp(lp(lp(x)))
10     yR = yL
11 }
```

`lp` and `lp3` are called *composite blocks* since they are themselves composed of other blocks. Line 9 contains three *instantiations* of `lp`. Composite block instances are

flattened during compilation, meaning that no set of blocks is ever treated as a black box.

3.1 Nested and Anonymous composite blocks

In order to keep the syntax flexible and coherent, the newly-introduced conditional branching syntax is based on other two additions, namely nested composite block definitions and anonymous composite block instantiations.

In the following code example,

```
1  y1, y2 = B1 (x) {
2      m = 0.5
3      t = B2 (z) {
4          t = z * m
5      }
6      y = {
7          y = B2(x)
8      }
9      y1 = y
10     y2 = -y
11 }
```

`B2` is a nested composite block that is visible only within `B1`. Scoping is hierarchical, so that code contained at an inner scope can reference variables and blocks defined in outer scopes: in the example, `m` is accessible within `B2`. At lines 6-8 an anonymous composite block is defined and instantiated at the same time. It has no name and no explicit inputs since it cannot be instantiated elsewhere. As a design choice, we opted for a full name masking strategy, meaning that identifiers (of both variables and blocks) in inner scopes shadow outer ones.

3.2 if-then-else

We introduced a syntax for conditional branching that is coherent with the declarative and modular nature of the language:

```
var1, var2, var3 = if (condition) {
    # anonymous block
    # executed if condition is true
    # var1, var2, var3 must be assigned
} else {
    # anonymous block
    # executed if condition is false
    # var1, var2, var3 must be assigned
}
```

In practice, each branch is implemented by instantiating an anonymous composite block. Please notice that it is not possible to define one branch only since variables, e.g., `var1, var2, var3`, must always be defined and the declarative nature of the language makes it impossible to define them elsewhere.

Perhaps the simplest use case for conditional branching is a 2x decimator. Here is an example implementation:

```
1  y = decimator(x) {
2      y, s = if (delay1(s)) {
3          y = x
4          s = 0
5      } else {
6          y = delay1(t)
7          s = 1
8      }
9      t = y
10     @s = 1
11     @y = 0
12 }
```

This turns a generic input sequence

$$x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}$$

into

$$x_1, x_1, x_3, x_3, x_5, x_5, x_7, x_7, x_9, x_9.$$

Here, `s` is used to hold the iteration state and `y` and `s` get updated differently according to the previous value of `s`. Note that, due to name masking, variable `t` is necessary to refer to the outer `y`: if line 6 were `y = delay1(y)`, then `y` on the right-side of the assignment would have referred to the internal-to-the-branch `y` and the block would generate an output sequence like $x_1, y_0, x_3, y_0, x_5, y_0, x_7, y_0, x_9$, where the initial value y_0 would have to be defined within the branch.

Indeed, it is possible to implement delays that are fully contained within a branch. As an example, consider the following naïve sawtooth generator:

```
1  y = saw_generator(enable, frequency) {
2      y = if (enable) {
3          phaseInc = frequency / fs
4          phase = frac(delay1(phase) + phaseInc)
5          @phase = 0
6          y = 2 * phase - 1
7      } else {
8          y = 0
9      }
10 }
```

Here, `phase` is updated only when `enable` is `true`.

3.3 New semantics and SDF

`decoder`, `switch` and `conditional delay` blocks do not explicitly appear in Ciaramella syntax, they are rather implicitly inferred and added to the IG as needed by the compiler. For example, figure 5 shows the SDF graph corresponding to the `decimator` example. It is important to note that there are 1 decoder (`DEC`) and 2 selects (`SEL`), and the same output from `DEC` is routed towards both `SEL`s. This originates from `y` and `s` being dependent on the same condition.

The scheduler checks whether delay-free loops are present in all potential execution paths and refuses to proceed if one is found. Since scheduling is performed after flattening, this arrangement fully preserves the modularity of the language.

4. DYNAMIC TO STATIC SCHEDULING

Let us consider, as a mere proof of concept, the following code:

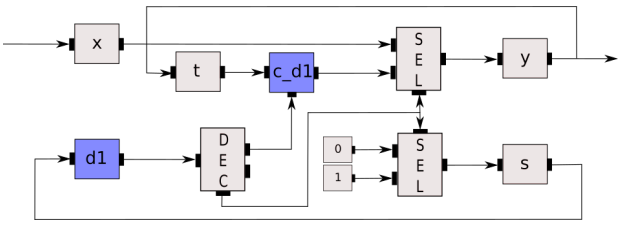

Figure 5: SDF graph corresponding to the `decimator` example

```
1  y1 = test(x) {
2      y1, y2 = if (x > 0.5) {
3          y1 = delay1(t1)
4          y2 = t2
5      } else {
6          y1 = t1
7          y2 = delay1(t2)
8      }
9      t1 = y2 + x
10     t2 = y1
11     @t1 = 0
12     @t2 = 0
13 }
```

By analysing both cases independently, it is possible to verify that the program does not contain delay-free loops and is always computable, yet no single scheduling order can be found that works in both cases. Indeed, when `x > 0.5` we end up having `t1` instantaneously depend on `t2` and viceversa otherwise.

More generally, such inversions of dependencies may arise in presence of an if-then-else block `B` when the following conditions are met:

1. `B` has multiple outputs (depending on the same condition);
2. there are one or more loops between the outputs and the inputs of `B`;
3. the same outputs of `B` rely on different inputs depending on the branch.

We have devised a strategy to avoid dynamic scheduling [4] in such scenarios. Zampogna discovers parts of the program meeting the conditions above and performs *if-then-else normalization*: the blocks outside the if-then-else block that are involved in loops get replicated and moved inside the branches, thus making such loops internal to each branch of `B` and invalidating condition 2. The following code contains the text representation of the graph resulting from if-then-else normalization in our example:

```
1  y1 = test(x) {
2      y1, y2, t1, t2 = if (x > 0.5) {
3          y1 = delay1(t1_)
4          y2 = t2
5          t1 = y2 + x
6          t2 = y1
7      } else {
8          y1 = t1
9          y2 = delay1(t2_)
10         t1 = y2 + x
11         t2 = y1
12     }
13     t1_ = t1
14     t2_ = t2
15     @t1_ = 0
16     @t2_ = 0
17 }
```

`t1 = y2 + x` and `t2 = y1` have been replicated and moved inside branches. According to scoping rules, `y1` and `y2` on the right-side of assignments do not refer anymore to the output of the implicitly-added select blocks, but rather to their inputs. `t1` and `t2` became if-then-else outputs and new intermediate variables (`t1_` and `t2_`) are added in order to keep scoping consistent. The resulting graph can be statically scheduled.

Moreover, this approach facilitates certain optimizations. Indeed, as parts of code are moved and replicated inside branches, the purely dataflow portion of the whole program tendentially increases, which simplifies data-flow analysis and SSA-based optimizations in modern compilers. On the other hand, the code size may increase, thus potentially affecting cache locality.

5. CONCLUSIONS

We addressed the problem of stateful conditional branching in SDF theory by proposing a simple addition which keeps the model flexible and coherent. Consequently, the Ciaramella DSP programming language has been augmented to support if-then-else constructs without compromising its fully declarative nature and its modularity. The Zampogna compiler can now handle such new features by transparently representing Ciaramella programs as SDF graphs and performing adequate scheduling and optimization.

The ambition of Ciaramella is to be capable of representing the largest class of audio DSP algorithms possible meeting the needs of both industry and amateurs. Introduction of conditional branching is an important step toward this goal, as it allows to both easily code new components, e.g. decimators and polyphase filters, and write optimized algorithms. Further research may be directed towards extending the proposed approach to implement ternary conditional operator for simple cases, *elseif* and *switch* constructs, as well as conditional loops. For the language to be feature complete, primitives like arrays, matrices, multirate support and n-delays are fundamental and they are still to be implemented.

6. REFERENCES

[1] P. Marrone, S. D'Angelo, F. Fontana, G. Costagliola, and G. Puppis, "Ciaramella: a synchronous data flow programming language for audio DSP," in *Proceedings of the 19th Sound and Music Computing Conference*, Saint-Étienne, France, June 2022.

[2] E. A. Lee and D. G. Messerschmitt, "Synchronous data

flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[3] ——, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, 1987.

[4] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[5] S. Ha and E. Lee, "Quasi-static scheduling for multiprocessor dsp," in *1991., IEEE International Sympoisum on Circuits and Systems*, 1991, pp. 352–355 vol.1.

[6] 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.

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

[8] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of FAUST," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.