Actors with stretchable access patterns

Ke Du, Stéphane Domas, Michel Lenczner

Actors with stretchable access patterns

Ke Du [a,b,*], Stéphane Domas [b], Michel Lenczner [b]

[a] *School of Science, Shandong Jianzhu University, Jinan 250101, China*
[b] *FEMTO-ST Institute (UMR 6174 CNRS), Univ. Bourgogne Franche-Comté (UBFC), Belfort 90000, France*

In this article, we propose a new framework based on dataflow graphs to abstract and analyze designs for hard-ware architectures. It is called Actors with Stretchable Access Patterns (ASAP). It can overcome some limitations of all Static Data Flow (SDF) based models like mandatory buffering between actors. This article details the fundamental contributions of ASAP. Firstly, it gives the definition of actors and their different patterns. It also illustrates the link between these notions and components written in VHDL through several examples. Secondly, it presents the main algorithms to check if a graph processes an input data stream correctly, which is called compatibility checking. Thirdly, it summarizes the principles of graph modification to enforce this correctness in case of some blocks are declared incompatible. Finally, it briefly describes our EDA tool called BlAsT which integrates the above principles, before presenting an application on a realistic FPGA design. It shows that ASAP overwhelms other models in terms of resources saving without any impact on the global latency. It also points out the ability of BlAsT to compute and to propose graph modifications and to generate the VHDL code of the whole design.

1. Introduction

On embedded systems, data processing applications that require high efficiency and throughput are more and more relying on hardware-based devices such as FPGAs and ASICs. But with the increase of the system size, it becomes harder and harder to manage designs manually. Therefore, model based design is a well adapted approach. Among the models, Data Flow (DF) has been extensively investigated and over the years, it has been expressed under several kinds of assumptions. As stated in 1987 at the beginning of the founding article [1]: "Data Flow is a natural paradigm for describing DSP applications for concurrent implementation on parallel hardware". This remark was done when the first FPGA emerged but thirty years later, even if FPGAs are far more powerful and the field of applications much larger than DSP, the same problem remains: how to build a hardware design by correct and efficient connection of functional blocks? Indeed, since hand coding a whole design in VHDL requires a great expertise and is a very long and tedious task, building it as graph of blocks that consume and produce data requires less efforts and knowledge, especially when a graphical tool (like Simulink) is available to create and to connect blocks. In most of specialized software packages, blocks are associated to real code and a tool is able to generate the code for the whole design. Nevertheless, a long process of simulation based on benchmarking is still necessary to validate the results produced by a graph of blocks.

Another solution based on dataflow is brought by high level languages like CAL [2]. It allows to describe a graph of *actors* that represent the functional blocks. Each time an actor is fired (or executed, triggered), it may consume and/or produce some data (*tokens*). It is also possible to specify conditions to fire the execution. Nevertheless, it is not possible to translate all CAL programs into VHDL (or Verilog). Subset of CAL like RVC-CAL [3] or similar ones like CAPH [4] solve this problem, mainly by setting constraints on the actor's description and assuming that they are linked with buffers. This is also the case of Floh language described in Ref. [5]. A Floh program is translated into a dataflow graph using a very small set of types of actor and buffer, presented in details in Ref. [6]. Each type is associated to its counterpart in Verilog which allows to easily produce synthesizable code. Nonetheless, all these solutions have more or less problems. For example, some propose to implement the link between actors as a FIFO but there is generally no methodology to estimate their size. If the designer chooses it too large, it constitutes a waste of resources. If it is too small, it produces deadlocks or incorrect computations. In Ref. [6], actors can be theoretically connected without buffers but a real design may produce deadlocks. Unfortunately, no principles are provided to decide if

* Corresponding author. School of Science, Shandong Jianzhu University, Jinan 250101, China.
E-mail addresses: duke@sdjzu.edu.cn (K. Du), sdomas@univ-fcomte.fr (S. Domas), michel.lenczner@univ-fcomte.fr (M. Lenczner).

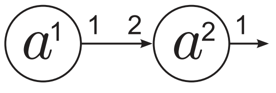

Fig. 1. An example of SDF graph with two actors.

a design produces deadlocks and to determine the optimal number and location of the buffers. There is also no guaranty that the code behaves as expected after synthesis, notably when it contains operations that have an architecture dependent time to complete. A typical example is an actor that uses the multiply operator to produce a result. Indeed, the actor's implementation must signal when valid outputs are produced so that the following FIFO avoids to store invalid values. It is generally achieved through a boolean enable signal. Assuming the multiply operator is translated as a * in VHDL, the following code snippet could be obtained after translation:

```
dout  <= op1 * op2;
dout_enb <= '1';
```

Unfortunately, it yields different behaviors depending on the FPGA architecture, the width of the operands and the choices of the synthesis tool. On the Spartan 6 used for our experiments, DSPs are 18 bits in width. Thus, as soon as an operand is larger than 18 bits, the result of the multiplication is available only after two or more clock cycles, and not just one. In this case, the enable signal is not synchronized with the valid output and the FIFO stores invalid values.

Prior to the definition of these languages, analyzing and enforcing the processing correctness in dataflow graphs thanks to models has been a highly investigated subject. Nevertheless, proposed approaches suffer from the same problems. They are based on the original one presented in Refs. [1,7] named Static/Synchronous Data Flow (SDF). They rely on the fact that the number of tokens consumed and/or produced during the execution of an actor and the execution time are fixed and known a priori. The graph in Fig. 1 shows two actors linked by a *channel* regarded as a FIFO. Each time a^1 is executed (or triggered), it produces one token. During its execution, a^2 consumes two tokens and produces one. It is worth noting that the duration of the executions is not taken into account.

In order to obtain a relevant behavior, all produced tokens must be consumed over a finite period of time. In Fig. 1, a^1 must execute twice as often as a^2, so that a^2 can consume enough tokens (two tokens of each execution polled from the buffer) for its own execution. This principle is called *the sample rate consistency* and it can be checked for any SDF graph. It represents a necessary but not sufficient condition to have FIFOs between actors that do not grow infinitely. The remaining problem is to determine when actors can start their execution. An obvious answer to reach maximal throughput is as soon as possible, as proposed by Geilen in the timed actor interface theory "*the earlier the better*" in Ref. [8]. Unfortunately, this proposition may also lead to infinite buffers. Nevertheless, it is possible to compute a valid and optimal schedule with finite ones by setting up assumptions like:

- Tokens are all produced at the end of the execution, in a single "shot", then stored in a buffer,
- An execution can start only if there are a sufficient number of input tokens in the buffer.

These assumptions yield a model quite distant from the behavior of real designs in VHDL. Indeed, output data are usually produced sequentially and sometimes in the middle of the execution. Furthermore, input tokens are generally consumed as soon as possible because waiting for them is a waste of time and resources.

Other variants have been proposed with different assumptions making them *self-timed scheduled* [9], such as CSDF [10,11], Heterochronous Data Flow (HDF) [12,13], Core Functional Data Flow (CFDF) [14], Scenario-Aware Data Flow (SADF) [15] and Static Data Flow with Access Patterns (SDF-AP) [16,17]. Synthesis and comparisons between these different approaches can be found in Refs. [18–21] but the latest one, SDF-AP, is taken as a reference in this article because it models the actor's behavior in a fashion close to that of real cores on FPGAs. Indeed, access patterns represent the pace of consumption and production of an actor. They are sequences of 1 and 0, with a length equal to the duration of the actor's execution. A 1 signals a clock cycle at which a token is consumed or produced. Compared with basic SDF approaches on real applications [22], it yields a reduction of buffer sizes and latency together with a possibly drastic increase of the throughput rate.

Nevertheless, SDF-AP suffers from limitations inherited from SDF principles. For example, buffering is mandatory but it could be avoided when delays are sufficient to synchronize several inputs of a single actor. Even if a delay is functionally similar to a FIFO, there is a complexity gap between their VHDL implementations and logic resources consumption. Source actors (i.e. actors with no inputs) are also a problem because they are taken into account in schedule computation. If the graph matches a real design, such actors would surely represent peripherals. But a lot of peripherals have a fixed (or nearly) execution schedule. Thus, this schedule is a constraint and cannot be freely set or computed. There is also a problem that is specific to SDF-AP because of the new constraints implied by patterns. Indeed, very simple designs lead to infinite buffer growth.

These limitations are discussed in details in the next section, and they inspired us to propose a new formalism called Actors with Stretchable Access Patterns (ASAP), that also relies on actors and patterns but with other definitions and usages. We assume that the data consumption rate of an actor is represented by a maximum value but it can vary under this value while conserving the correctness of the production. The notion of rate variability already exists in the Variable Rate Data Flow model (VRDF) [23] but with another definition. Indeed, VRDF assumes that an actor may consume (and produce) more or less data during each execution, which, over time, effectively yields a rate variability. In ASAP, the volume of consumed data never changes between executions. The rate variability expresses the fact that the consumption rate of an actor is directly determined by the production rates of its precursors in the graph and if they are lower than the actor's maximum, its production is still correct though slower. This constraints actor's implementation but does not increase its coding complexity, as shown in Section 3.1.7. ASAP represents the maximum consumption rate by a pattern that is only theoretical. A real execution pattern may be a stretched version of the theoretical one. This constitutes the origin of the expression "Actors with Stretchable Access Patterns". Regarding channels implemented as FIFOs and controllers, they are highly resource consuming. Thus, we replace the buffer size optimization approach by an analysis of the real actor's patterns considering the graph without any buffer. If the analysis detects a problem, we will search for the minimum set of delays, decimators and if needed, buffers to be added between some actors so that the graph processes data produced by the sources correctly. To summarize, our contribution is not only a new model but a whole framework with the three main features:

- A model for actor's executions that allows to take a wider range of behaviors into account than previous models,
- A static analysis procedure that allows to detect if a design processes input streams correctly, without simulations,
- A modification procedure of the design when correctness is not reached.

This approach serves as a basis of an EDA tool called BlAsT (Block Assembly Tool). It integrates all the above principles to help non-expert users in producing FPGA designs.

The rest of the paper is structured as follows. Section 2 recalls the SDF-AP model properties and behaviors before illustrating some of its limitations through simple examples. The novel model of actors is elaborated in Section 3, where their properties and their patterns

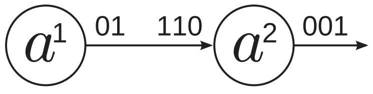

Fig. 2. An example of SDF-AP graph with two actors.

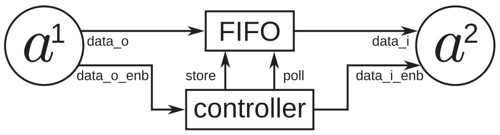

Fig. 3. The general structure (FIFO + controller) to interconnect two actors in SDF-AP.

are discussed in details. Examples illustrate the fundamental algorithms related to the model. Section 4 presents a case to introduce the principles of graph modifications yielding correctness. It must be noticed that the whole procedure is complex and requires too much detailed explanations to fit in the present article. So, only the main lines are given. Section 5 presents a realistic test case that points out the efficiency and advantages of ASAP compared with SDF-AP. It also presents some elements of BlAsT. Finally, we draw the conclusions and perspectives in Section 6.

2. The SDF-AP model

2.1. Principles

Static Data Flow with access patterns is a SDF based model using additional *access patterns* to describe the clock cycles at which tokens are produced or consumed by actors during their executions [16,17]. Both kinds of patterns are sequences of 1 and 0, and are called **consumption pattern** and **production pattern** respectively. Their length is equal to the duration of the actor's execution. Fig. 2 shows an example of SDF-AP graph, based on the example in Fig. 1. Each time a^1 is triggered, its execution lasts two cycles. It produces nothing during the first clock cycle and one token during the second one. When a^2 is triggered, its execution lasts three cycles. It consumes one token at each clock cycle of the first two and produces one at clock cycle 3.

It is worth noting that consumption pattern must be matched strictly to ensure the result correctness. In this example, if a valid data is presented on the input of a^2 at the third clock cycle of its execution, or if there are no valid data at clock cycle 1 or 2, the actor will produce incorrect results. To enforce this constraint, as in other SDF based models, SDF-AP assumes that the channel between two actors is a buffer. In Refs. [16,24], the same group of authors propose a realistic representation (from the hardware point of view) of these buffers: a FIFO with a controller that manages the store and poll requests. This general structure is shown in Fig. 3. The actor's implementation must define additional ports `data_o_enb` and `data_i_enb` to manage FIFO accesses. Generally, a store request is issued as soon as a token is produced by a^1 on `data_o`, that is when `data_o_enb` is asserted. Poll requests occur at clock cycles computed by the access patterns and the scheduling. In order to make the global latency shorter and to minimize the size of the buffer, tokens must be polled as soon as possible but in a sequence that matches the consumption pattern of a^2.

To illustrate this behavior, we take the design in Fig. 2 and assume that a^1 produces a result every two clock cycles. The couple FIFO/controller is supposed to behave as a VHDL implementation synchronized on a global clock. It means that for a poll request at clock cycle t, the result is available on the FIFO's output at clock cycle $t+1$ (same with store requests). Under these conditions, the SDF-AP model implies the following schedule:

```
entity simpleCoreSDFAP is
  port(
    clk       : in  std_logic;
    reset     : in  std_logic;
    d1_in     : in  std_logic_vector(7 downto 0);
    d2_in     : in  std_logic_vector(7 downto 0);
    start     : in  std_logic;
    d_out     : out std_logic_vector(7 downto 0);
    d_out_enb : out std_logic );
end simpleCoreSDFAP;
```

Fig. 4. An example of entity compliant with SDF-AP.

- At $t = 2$: a^1 produces its first token and the controller issues a store request.
- At $t = 3$: the first token is available in the FIFO.
- At $t = 4$: a^1 produces its second token and the controller issues a store request. It also issues a poll request.
- At $t = 5$: the second token is available in the FIFO and the first token is available on `data_in` of a^2. This latter starts its execution and consumes it. The controller issues a poll request.
- At $t = 6$: the second token is available on `data_in` of a^2, that consumes it. a^1 produces its third token and the controller issues a store request.
- At $t = 7$: the third token is available in the FIFO and a^2 produces its result.
- …

This leads to an execution of a^2 every four clock cycles, the first being at $t = 5$ and a minimum size of 2 for the FIFO. With a SDF model, the throughput would be the same but with the first execution at $t = 6$ (or even more depending on the access policy of the buffer) and a minimum buffer size of 4. The gain is moderate but for actors consuming hundreds of tokens, it may be huge [16,22].

It can be noticed that it is theoretically useless to associate an enable signal `data_o_enb` to data signal `data_o`. Indeed, once the clock cycles at which a^1 is triggered are known, the store requests schedule can be determined from its production pattern. Nevertheless in practice, it is simpler to use such an enable signal because the store requests are simply driven by its assertion to `true`. Similarly, `data_i_enb` is useless since a^2 is supposed to follow a fixed consumption policy given by its consumption pattern. But there must be at least one input signal that triggers its execution. The FIFO controller is in charge to emit this signal and the poll requests so that a^2 effectively receives valid values at the correct clock cycles according to its consumption pattern. Under these conditions, patterns can be deduced directly from the code of the cores. In order to illustrate these remarks, Fig. 4 describes a VHDL entity called `simpleCoreSDFAP` with two inputs and one output port.

Assuming that the architecture part of `simpleCoreSDFAP` contains the code given in Fig. 5, the consumption pattern is $\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$, the production pattern is $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ and the execution time is equal to 3 clock cycles. Indeed, it stays in `idle` state until `start` is asserted to 1. This condition triggers the execution of the actor, the consumption of values received on both inputs to compute the result of function `fun1`. It corresponds to the first column of the consumption pattern (containing only 1). Then, state changes to `step1`. During the next clock cycle, the actor only consumes the value on the first port to compute the result of `fun2` that is assigned to `d_out`. It corresponds to the second column of the consumption pattern. Then, it returns to the `idle` state and notifies that the final result will be available at the next clock cycle by asserting `d_out_enb` to 1. It yields an execution time of 3 clock cycles.

```
compute : process (clk,reset)
  begin
    if reset = '1' then
    state <= idle;
      ...
    elsif rising_edge(clk) then
      d_out_enb <= '0';
      if state = idle and start = '1' then
        res <= fun1(d1_in,d2_in);
        state <= step1;
      elsif state = step1 then
        d_out <= fun2(res,d1_in);
        d_out_enb <= '1';
        state <= idle;
      fi
    fi
  end process compute;
```

Fig. 5. An example of architecture compliant with SDF-AP.

This example clearly shows that both functions are called even if the validity of received data does not strictly match the consumption pattern, yielding an incorrect result. As said above, the FIFO controller prevents such dysfunction.

It can also be noticed that two subsequent executions may overlap: the second one could be triggered at the third clock cycle of the first one while it produces its result.

2.2. Limitations

As described above, the principles of SDP-AP model are those of SDF enriched by the concept of pattern. Even if it brings a major improvement, the underlying assumptions yields a waste of resources and a restricted representation of core behaviors. This section describes the three main limitations inherent to these assumptions and introduces potential solutions.

2.2.1. Strict pattern conformance and buffering

This first limitation has already been described using Fig. 3. It comes from the condition that actor's consumption patterns must be matched strictly for actors to produce correct results. In the best case, when the production pattern of an actor is equal to the consumption pattern of the next actor, no buffering is needed. But in the general case, FIFOs with controllers must be used.

In the illustrating example shown in Fig. 6, a source emits frames of 16 data at each execution followed by a decimator that keeps one data out of two to feed an average filter with a mask of size 3. Patterns use the standard regular expression syntax to specify groups (with parenthesis) and repetitions (with embraces). For example, (01){8} means that 01 is repeated 8 times. Assuming that the source execution starts at clock cycle 1, the decimator produces data at clock cycles $2n$ ($n \in \mathbb{N}^*$). Applying a strict pattern conformance, the filter must consumes 8 data during 8 consecutive clock cycles. The SDF-AP solution is to store a certain amount of data in a FIFO after the decimator before triggering the filter execution. In this example, the earliest time to start the filter is at clock cycle 11 with a FIFO size of 4 (or 5 depending on the priority between store and poll requests). More generally, for a decimator producing N data, the filter can start its execution at clock cycle $N+3$ and the buffer size is $\lceil \frac{N}{2} \rceil$.

This behavior is not a problem in itself but yields a waste of resources. Indeed, buffering can be totally avoided if the filter is able to consume data without strictly matching its consumption pattern, that is only when the decimator produces data. In terms of implementation, this represents minor changes in the code of cores as shown in Section 3.1.7.

2.2.2. Auto-concurrency

In Ref. [17], authors use the notion of *auto-concurrency* in SDF-AP graphs to describe the fact that "multiple instances of an actor can execute simultaneously". They define a parameter ii that represents the **minimum** number of clock cycles between two executions of the same actor. They also add that: "this may be not feasible in practice due to restrictions like finite resources, IP properties, etc.". Despite the fact that they do not give precisions about what they consider to be a real auto-concurrent actor (at the implementation level for example), we notice that actors using a sliding window on input data are in this case.

To point out the limitation with such actors, we take the following example: an 1D average filter with a mask of size 3. Denoting its input data as: d_1, d_2, d_3, …, it produces $\frac{d_1+d_2+d_3}{3}$, $\frac{d_2+d_3+d_4}{3}$, …, where, for simplicity, we omit the averages at bounds.

If the filter has an "internal" knowledge on its processing end, the definition of ii is relevant. For example, if the filter is implemented to operate on sequences of 5 data, its consumption pattern is $\begin{bmatrix}1 & 1 & 1 & 1 & 1 & 0\end{bmatrix}$ and its production pattern is $\begin{bmatrix}0 & 0 & 0 & 1 & 1 & 1\end{bmatrix}$ (N.B.: this assumes that the last sum and the division by 3 can be done in a single clock cycle). In this case, the actor cannot execute once again before five inputs have been consumed (i.e. $ii \geq 5$), otherwise it would produce incorrect results. This is a weak auto-concurrency since the overlap occurs only when all needed input data have been consumed. Thus, the same input data is not used by several concurrent executions.

The problem arises when the processing end is driven externally. For example, there may be a boolean input that is asserted to specify that end. Expressing a pattern with an unpredictable length is not a relevant solution. Auto-concurrency allows to express it for a single average. For example, if we assume that the filter is able to compute the sum and the division by 3 in a single clock cycle, its consumption pattern is $\begin{bmatrix}1 & 1 & 1 & 0\end{bmatrix}$ and its production pattern is $\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}$.

In order to compute a sequence of averages correctly, this filter **must** execute once again as soon as there is a valid input data. This yields strong concurrency since the same input data will be used for three concurrent executions (except at the bounds). With such an actor, $ii = 1$ is the single possible value, and not a minimum. In conclusion, auto-concurrency, as defined in SDF-AP model, is inadequate and another definition must be given to match a larger range of actor's behavior.

2.2.3. Infinite buffering

The third limitation comes from the combination of the two previous ones and is illustrated by the example given in Fig. 7. The design is

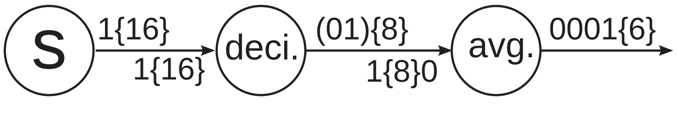

Fig. 6. A decimator connected to an average filter with fixed size data flows modeled by SDF-AP.

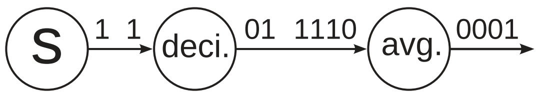

Fig. 7. A decimator connected to an average filter with an infinite data flow modeled by SDF-AP.

functionally equivalent to the previous example, but actors operate in a pipelined way, on an infinite data flow. In this case, patterns must be expressed for a single decimation and average, with actors triggered at each clock cycle. Since patterns must be strictly matched in SDF-AP model, it also implies that the filter must be fed at each clock cycle with a valid data. Unfortunately, no matter what the rhythm of source production is, the decimator will never produce data at contiguous clock cycles. The single solution is to put a FIFO after the decimator with an infinite number of data before polling them, which is impossible.

Similarly to the previous one, the problem with an infinite buffer can be solved by modifying the filter to consume data only when they are available.

Based on the above analysis, interesting propositions can be done to limit resource consumption and to obtain a valid schedule on a larger class of designs. They mainly rely on a new definition of the auto-concurrency property, an optional buffering and a relaxed pattern conformance. This leads to our model that is presented in details in the next section.

3. The ASAP model of actors

Even if ASAP aims to overcome the limitations of previous models, it keeps the principle of static analysis that implies some unavoidable constraints. For example, some actors have a variable behavior because it depends on the consumed values. A one-to-N switch with an input that selects which output is chosen is a perfect example of such an actor. Under some conditions, it is possible to conduct a static analysis, as proposed in Refs. [25,26]. Nevertheless, it is not possible in the general case and it implies the use of simulations to retrieve the behavior under specific execution conditions. This is why we consider, in the following, only actors with a behavior independent from input values.

As mentioned in the introduction, our final objective is to integrate our model and its related algorithms in a software tool that allows to create functional FPGA designs. Thus, ASAP has been designed to be applicable to new cores development (for example in VHDL). We have also taken the possibility to model existing cores into account provided they follow some simple constraints. Since there are many ways to implement a functionality, we have defined pattern properties to encompass a large range of behaviors, even those that could be considered as inefficient or nearly useless in practice. Moreover, as with many models, the possible results are wider than what can be used in real designs. This is why ASAP allows some cases that are unrealistic or even unfeasible in term of implementation.

Even though we have a practical goal, the following sections presents ASAP from a theoretical point of view, as in the majority of related works about models. For example, unfeasible cases are determined and explained. Some examples of patterns may be considered as unrealistic in practice but we use them to present some particularities of the model and their implications. It is also the case for algorithms that are not constrained by realistic assumptions. Nevertheless, some propositions of implementation and practical examples will be given to help the reader to catch the link between the model and our final objective.

Section 3.1 describes an actor considered alone and Section 3.2 discusses about a graph of actors.

3.1. *Actor's structure and behavior*

ASAP relies on a very simple structural constraint for actors and a small set of notions to describe their behavior. The following sections give their theoretical definition and a tiny illustration. The tight relations between these notions are explained in Section 3.1.6. Finally, Section 3.1.7 presents some examples to illustrate these notions and to describe their link with real implementations in VHDL.

3.1.1. *Clock and ports*

We assume that an actor is synchronized on a global clock signal that is always enabled. Clock cycles are numbered from 1 to ∞. When the execution of an actor is triggered at a given clock cycle, it consumes and produces data on its ports. An actor has $P_I \in \mathbb{N}^*$ input ports and $P_O \in \mathbb{N}^*$ output ports. Actors without inputs or outputs are called *sources* or *sinks* respectively.

Each input and output port corresponds to a couple of signals (`data, validity`). The type of `data` is left undefined and `validity` is a boolean signal. An **input group** is the set of values of the `data` signal received on the inputs, at a given clock cycle. Similarly, an **output group** is for the outputs.

It is worth noting that existing cores may already match these constraints, as those that comply with the AIX4-stream protocol in non-blocking mode. Thus, such cores can also be modeled with ASAP.

3.1.2. *Execution and concurrency*

An actor is **self-triggering**. For source actors, their first execution starts at clock cycle 1 of the global clock. A new execution is automatically triggered at the clock cycle following the end of the current one. For other actors, their execution is triggered by the first input group with validity values that conform to a particular state. For example, in Fig. 9, both inputs must be valid to trigger the execution. The execution is considered to be completed when the last result has been produced.

Moreover, depending on how the actor is implemented, it can be triggered again even if it is already executing, which yields **concurrent executions**. The most extreme case is when a new execution starts while the previous has not totally consumed all needed groups. In that case, if different executions consume the same input group, it yields **concurrent consumptions**. These notions are essential to model the behavior of actors that process input streams in a pipelined way. The average filter mentioned in Section 2.2.2 that operates on a sliding window is in that case.

3.1.3. *Consumption*

Since an actor is able to wait, its execution time is not fixed. It depends on the actors that produce the data to be consumed. Nevertheless, there exists a minimum execution time corresponding to the fact that the actor never has to wait and that guaranties of correct results. The **consumption pattern** (*CP*) represents the consumption policy of all input ports at each clock cycle of the minimal execution time. It is quite similar to the one defined in SDF-AP model but it takes the auto-concurrency into account. For convenience, it is expressed as a matrix but should be interpreted as a sequence of column vectors. The *t*th column describes the consumption policy of the actor at the clock cycle *t*, relatively to the beginning of its execution. If L_{CP} is its length, then:

$CP = [CP_{i,t}]$, for $i \in \{1, \ldots, P_I\}, t \in \{1, \ldots, L_{CP}\}$, $CP_{i,t} \in \{1, 0, \times\}$, and

- $CP_{i,t} = 1$, if the actor must consume the data on the input i to compute a correct result,
- $CP_{i,t} = 0$, if the actor does not need the data for the current execution, but a next execution may consume it,
- $CP_{i,t} = \times$, if the actor must not consume the data for any current execution.

The $\times$ corresponds to the 0 used in to SDF-AP. The 0 is a new notion introduced to represent certain actors with concurrent consumptions. Together with the trigger delay notion presented in the following section, they are necessary to obtain a correct behavior analysis of such actors when they are used within a graph. An example is given in Section 3.1.7.

If the tth column contains one or several 1, it means that the actor consumes at least one data. It is called a **valid column** because it corresponds to a set of data for which at least one validity signal is equal to 1 (NB: this notion is also used for the other types of patterns defined in ASAP). By extension, the associated set of data is called a **valid input group**. V_{CP} represents the number of valid columns in CP, i.e. the number of valid input groups that must consumed to complete an execution.

For instance, $CP = \begin{bmatrix} 1 & \times & 1 & 0 & 0 & 0 & 1 \\ 1 & \times & 0 & 0 & 1 & 0 & \times \end{bmatrix}$ corresponds to $P_I = 2$, $L_{CP} = 7$, and $V_{CP} = 4$. It presents a possible (even though unrealistic) CP with different combinations of 1, 0 and $\times$ in a single column.

3.1.4. *Triggering delay*

The delay between two executions of an actor is noted as Δ. It corresponds to the number of valid input groups (not the clock cycles as in SDF-AP) that must be consumed by an actor before it starts another execution. Consequently, $1 \le \Delta \le V_{CP}$. From the consumption point of view, Δ is a fixed value. Thus, if the $\Delta + 1$ valid input group has validity values that do not conform to the first column of CP, the next execution will not start and the actor will eventually produce incorrect results.

3.1.5. *Production*

The **production pattern** (PP) is the counterpart of CP but for results. The tth column of PP describes if the actor produces valid results or not on its outputs at clock cycle t relatively to the beginning of its execution. If L_{PP} is its length, then:

$PP = [PP_{o,t}], o \in \{1, \dots, P_O\}, t \in \{1, \dots, L_{PP}\}$, $PP_{o,t} \in \{1, 0\}$, and

- $PP_{o,t} = 1$, if the actor produces a valid result,
- $PP_{o,t} = 0$, otherwise.

V_{PP} represents the number of valid columns (i.e. with at least a 1 in the column) in PP.

Moreover, an actor produces results in a logical order, that is as soon as possible after a sufficient number of valid groups have been consumed. The **production counter** (PC) represents this property. It defines the number of valid input groups that are necessary to produce a valid output group. It is expressed as a vector $PC = [PC_o]$, with $o \in \{1, \dots, V_{PP}\}$ and $PC_o \in \{1, V_{CP}\}$. It is also assumed that if $i < j$ then $PC_i \le PC_j$.

For instance, $PP = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$ corresponds to $P_O = 2$, $L_{PP} = 6$, and $V_{PP} = 3$. A possible production counter could be $PC = \begin{bmatrix} 2 & 4 & 5 \end{bmatrix}$

3.1.6. *Origins and relationships between these notions*

Some definitions given above, notably PP, are quite obvious because they match the behavior of components implemented on hardware-based devices like FPGAs. Other definitions require clarifications to be fully understand.

Concurrent executions are an abstraction. They do not corresponds to a physical reality implemented, for example, with several identical VHDL processes that run concurrently. They are used to represent a pipelined execution of an actor. In the most common case, such an actor consumes data at each clock cycle to initiate the first stage of the pipeline, which triggers a new execution. Depending on the actor's task and the number of concurrent executions, this input group may also be

```
entity simpleCoreASAP is
  port(
    clk         : in  std_logic;
    reset       : in  std_logic;
    d1_in       : in  std_logic_vector(7 downto 0);
    d1_in_enb   : in  std_logic;
    d2_in       : in  std_logic_vector(7 downto 0);
    d2_in_enb   : in  std_logic;
    d_out       : out std_logic_vector(7 downto 0);
    d_out_enb   : out std_logic );
end simpleCoreASAP;
```

Fig. 8. An example of entity compliant with ASAP.

used in other stages of the pipeline, which yields concurrent consumptions. Nevertheless, ASAP actors are able to wait for valid input groups. Thus, there may be several clock cycles between the triggering of two concurrent executions. This justifies definition of Δ, based on a number of consumed groups and not clock cycles. In practice, a pipelined actor has generally $\Delta = 1$, which corresponds to a new execution each time a valid input group is consumed. But ASAP allows other values. It can be noticed that $\Delta = V_{CP}$ corresponds to an actor without pipeline.

PC allows to differentiate the behavior of actors that have the same consumption and production patterns. For example, an average filter with a mask of size 3 and a threshold filter that operate on a sequence of 5 values have:

$CP = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}, PP = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$.

Nevertheless, they have a different production counter:

$PC^{thre.} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}, PC^{avg.} = \begin{bmatrix} 2 & 3 & 4 & 5 & 5 \end{bmatrix}$.

This differentiation is necessary when such actors are used in a graph and they have to wait for some input groups during their execution. For example, assuming that the second input group is delayed, it has no influence on the threshold filter to produce its first result. It is not the case for the average filter because it needs two input groups to produce its first result.

There are also relationships between all these notions. In practice, CP, Δ, PP and PC are given by the actor's implementation, as shown in the next section. But in purely theoretical examples, it is impossible to choose their value freely.

Firstly, there may be impossible combinations of Δ and CP. For instance, $CP = \begin{bmatrix} 1 & \times & 1 \\ 1 & 1 & 0 \end{bmatrix}$ and $\Delta = 1$ are inconsistent. Assuming that the first data group triggers the first execution at clock cycle t, since $\Delta = 1$, the second data group automatically triggers the second execution at $t + 1$. For that execution, the actor must consume a valid data on both inputs to produce a correct result. However, for the first execution, the consumption policy is given by the second column of CP with an $\times$ that forbids consumption on input 1 for any execution. These two constraints are in contradiction thus Δ cannot be equal to 1 (but 2 and 3 are possible).

Secondly, certain combinations of PP, PC, CP and Δ are forbidden, as for example, $CP = \begin{bmatrix} 1 & 1 \end{bmatrix}, \Delta = 1, PP = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$ and $PC = \begin{bmatrix} 2 & 2 \end{bmatrix}$. Indeed, since $\Delta = 1$, two successive executions yield a result at the same clock cycle, which is not physically feasible. Nevertheless, $\Delta = 2$ or $CP = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}$ would be correct.

3.1.7. *Illustrative examples*

In order to illustrate the notion of ports, Fig. 8 describes an entity called `simpleCoreASAP` with two inputs and one output port. In this example, `d1_in` and `d2_in` represent the `data` signals of the input port, while `d1_in_enb` and `d2_in_enb` the `validity` signals. The same principle applies for the output.

```
consumption : process (clk,reset)
  begin
    if reset = '1' then
      state <= idle;
      ...
    elsif rising_edge(clk) then
      d_out_enb <= '0';
      if state = idle and d1_in_enb = '1'
                      and d2_in_enb = '1' then
        res <= fun1(d1_in,d2_in);
        state <= step1;
      elsif state = step1 and d1_in_enb = '1'
                          and d2_in_enb = '0' then
        d_out <= fun2(res,d1_in);
        d_out_enb <= '1';
        state <= idle;
      fi
    fi
  end process consumption;
```

Fig. 9. An example of architecture compliant with ASAP.

When the `validity` signal is asserted to be 1 on some ports, the associated `data` signals carry a value that **must** be consumed and used for the current actor's execution. If the actor is not ready to consume them, the data will be lost and the actor will produce incorrect results. When the `validity` signal is asserted to be 0 on a given port, the actor **must not** consume the value on `data`. Nevertheless, it may be in a state where it really needs a value on that port and maybe more. In that case, the actor **must be able to wait** an unbounded number of clock cycles until the `validity` signal becomes equal to 1 for all these ports.

```
consumption : process (clk,reset)
  begin
    if reset = '1' then
      state <= idle;
      ...
    elsif rising_edge(clk) then
      d_out_enb <= '0';
      if state = idle and d_in_enb = '1' then
        res1 <= f(d_in);
        state <= init;
      elsif state = init and d_in_enb = '1' then
        res2 <= g(res1);
        -- initiate a new exec.
        res1 <= f(d_in);
        state <= stable;
      elsif state = stable and d_in_enb = '1' then
        -- finalize prev. prev. exec.
        d_out <= h(res2,d_in);
        d_out_enb <= '1';
        -- continue the prev. exec.
        res2 <= g(res1);
        -- initiate a new exec.
        res1 <= f(d_in);
      fi
    fi
  end process consumption;
```

Fig. 10. Illustrative example 1.

```
consumption : process (clk,reset)
  begin
    if reset = '1' then
      state <= idle;
      ...
    elsif rising_edge(clk) then
      d_out_enb <= '0';
      if state = idle and d_in_enb = '1' then
        res1 <= f(d_in);
        state <= step1;
      elsif state = step1 then
        res2 <= g(res1);
        state <= step2;
      elsif state = step2 and d_in_enb = '1' then
        d_out <= h(res2,d_in);
        d_out_enb <= '1';
        -- initiate a new exec.
        res1 <= f(d_in);
        state <= step1;
      fi
    fi
  end process consumption;
```

Fig. 11. Illustrative example 2.

Fig. 9 illustrates these properties. It is an example of VHDL code that could be found in `simpleCoreASAP`. In a core compliant with ASAP, the execution is always triggered by a condition on the validity of inputs but not by a dedicated signal as in SDF-AP. In this example, the core stays idle until a valid data is received on both input ports. Then, it consumes these data to compute something and passes in `step1` state. Contrarily to SDF-AP, it can stay indefinitely in that state, until another condition on the input validity is met. In this case, it waits a valid data only on the first port. Then, the actor consumes it to compute the final result. After that, it returns to the `idle` state. Consequently, if the consumption conditions are not respected, some input data will be lost and the actor will either not start its execution or compute correctly. Such a case is discussed in Sections 3.2.2 and 3.2.3.

As mentioned above, patterns and related notions are deduced from the actor's implementation. For the code in Fig. 9, it is very easy to determine the characteristics of the output. Indeed, a single port produces a single result at clock cycle 3 of the execution, after two input groups have been consumed. It gives $PP = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$, $PC = \begin{bmatrix} 2 \end{bmatrix}$, $P_O = 1$, $L_{PP} = 3$, and $V_{PP} = 1$.

Δ is also quite straight forward to determine. An execution can start only if the state is `idle`. Since the process returns to `idle` only if two input groups have been consumed, it implies that $\Delta = 2$. It also implies that there are no concurrent consumptions.

Finally, *CP* can be deduced from the conditions on validity signals and changes of state. Since it is possible to change of state at each clock cycle, it gives $CP = \begin{bmatrix} 1 & 1 \\ 1 & \times \end{bmatrix}$, $P_I = 2$, $L_{CP} = 2$, and $V_{CP} = 2$. It is worth noting that the $\times$ represents the condition `d2_in_enb` = '0', which forbids to consume a data on the second port when state is `step1`. Nevertheless, since there are no concurrent consumptions, a 0 would be equivalent.

This assertion raises the question of the usefulness of $\times$ in consumption patterns. Actually, they are necessary when there are concurrent consumptions and the following examples present a proof of this. Figs. 10 and 11 contain VHDL codes that carry out the same functionality but implemented in two different ways. They use three functions `f`, `g` and `h` to produce a single result from two inputs that are consumed at most every two clock cycles. In the following, *a* is the actor that models the first code, and *b* the second.

From the code in Fig. 10, we deduce that $\Delta^a = 1$ because a new execution starts after each consumption. CP^a is easy to deduce from the very first execution. As soon as there is a valid input, there are two consumptions separated by at least two clock cycles. It implies that $L^a_{CP} = 3$ and $CP^a_0 = CP^a_2 = 1$. Determining CP^a_1 is also easy because at the `init` state, a waits for a valid input that is not used for the first execution but that is consumed to trigger the second one. Thus, by definition $CP^a_1 = 0$, yielding $CP^a = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}$. Finally, $PP^a = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$ and $PC^a = 2$ because the result is released after the consumption of two groups at the fourth clock cycle.

From the code in Fig. 11, we deduce that Δ^b, L^b_{CP}, CP^b_0, CP^b_2, PP^b and PC^b have the same values as those of a. A new execution starts after each consumption and the result is produced from two consumptions. Nevertheless CP^b_1 is different. When the process is in the `step1` state, no input is consumed by any execution. If a^2 is used within a graph and receives a valid data while it is in this state, the data will be lost and the graph will produce incorrect results. The graph analysis must be able to detect such a case. If $C_1 = 0$, the analysis would consider that the data may be used by another execution and would conclude that there is no problem. If $C_1 = \times$, the presence of a valid data during the `step1` state is considered to be a faulty case by the analysis.

It is worth noting that a is able to execute and to produce a result at each clock cycle whereas it is at most every two clock cycles for b. Consequently, the question of the usefulness of taking the case of b into account could be raised. But as said previously, ASAP does not rely on such a criterion but on the fact that it can model the largest number of behaviors.

3.2. *The graph of actors and its analysis*

As in SDF based models, a design is abstracted by a connected and oriented graph of actors. The main constraint is that the graph must not contain cycles. Such a case would prevent to apply the analysis principles presented below. Indeed, for a given actor a, it is necessary to know what is sent by its direct predecessors to determine what it produces for its direct successors. A cycle towards a implies that one of its inputs comes from one of its successors. Thus, it would be impossible to compute what it produces.

3.2.1. *Input and output patterns*

Since the first clock cycle of the global clock, source actors execute repeatedly and produce data groups that must be consumed by their successors in the graph. These successors are themselves producing data groups that must be consumed by their own successors, and so on until actors with no successors. Thus, the pace at which a given actor receives valid input groups is given by the pace of production of its predecessors and not by CP. This pace is called the **input pattern** (IP). It corresponds to the variations of the validity signal received by the input ports and is represented by a matrix of 0 and 1.

Assuming we only consider global clock cycles ranging from 1 to T, the input pattern is noted

$IP = [IP_{i,t}], i \in \{1, \ldots, P_I\}, t \in \{1, \ldots, T\}$, where $IP_{i,t}$ is the value of the `validity` signal received by the input i at the clock cycle t.

The pace of production is called the **output pattern** (OP). It corresponds to the variations of the validity signal emitted by the output ports and is noted

$OP = [OP_{o,t}], o \in \{1, \ldots, P_O\}, t \in \{1, \ldots, T\}$, where $OP_{o,t}$ is the value of the `validity` signal produced by the output o at the clock cycle t.

For instance, $IP = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$ is for an actor with 2 inputs ($P_I = 2$) and $T = 8$ clock cycles, and $OP = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$ is for an actor with 3 outputs ($P_O = 3$) and $T = 6$ clock cycles.

3.2.2. *Admittance pattern*

An additional notation is necessary for patterns, notably in the following algorithms. For a given type XP of pattern, $XP_{*,t}$ represents the state of `validity` signal for the data group at clock cycle t, which is the tth vertical vector in XP.

The distinction between CP and IP is at the origin of the word "stretchable" in ASAP. Indeed, considering a single execution of an actor a without concurrent consumption, we can determine the part of IP that encloses the valid input groups received by a during this execution. If that part corresponds to CP exactly, a produces correct results by definition. If it is not the case, a may still produce correct results if that part is a stretched version of CP, i.e. with additional null columns. Since these columns represent invalid groups and a is able to wait for valid input groups, they have no impact on the execution correctness. For example, in the case of the average filter in section 2.2.3, $CP = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$ but if it receives $IP = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$ from the decimator, it can work as well.

IP is considered to be **compatible** with a consumption pattern CP if all the actor's executions implied by IP produce correct results. Due to the structure of CP and the value of Δ, checking that compatibility may turn out to be a non-trivial task in practice, notably when there are concurrent consumptions. The **admittance pattern** (AP) presented below is at the heart of the solution provided by the **compatibility check algorithm** detailed in Section 3.2.3.

The simplest case is when there are no concurrent consumptions. Then, AP is simply constituted of several concatenations of CP. If removing some of the null columns of IP leads to AP, then IP is compatible with CP. For instance, for $P_I = 2$, $L_{CP} = 4$, $V_{CP} = 3$, and $\Delta = 3, IP = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$ and $CP = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$ are compatible.

It is obvious that removing columns $1, 2, 4, 6, 9, 12$ leads to the concatenation of two CP.

In case of concurrent consumptions, checking the compatibility requires a similar process of removing null columns and matching with a reference. Nevertheless, this problem is more complex because it raises up some questions about the number of compatible patterns and their structures.

For a given number of complete executions n_{exe}, there exists at least one admittance pattern. For a better understanding, the principles of its construction are firstly described with a basic case in Fig. 12 and Example 1, and further exposed in details in Algorithm 1.

Example 1. $CP = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$, $P_I = 2$, $L_{CP} = 3$, $V_{CP} = 3$, $\Delta = 1$, and $n_{exe} = 4$.

In Example 1, the admittance pattern is built by copying n_{exe} times the pattern CP on a graph where the columns represent the clock cycles and the rows represent the sequence of executions. Assuming that the first execution starts at the clock cycle 1 and that the data groups are available as soon as possible, we can copy CP in the first row, first column. Since $\Delta = 1$, a new execution is triggered by each valid data group. Thus, we copy CP in the second and third rows, respectively at columns 2 and 3. This copy process is done n_{exe} times by determining for each execution when it must (or can) start to compute correct results.

Even if there are different consumption policies at each clock cycle for concurrent consumptions, they can be unified by doing a logical

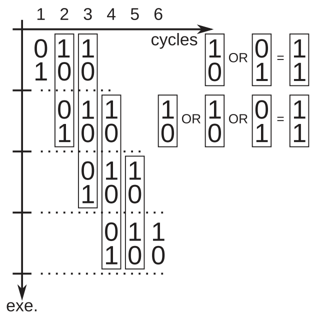

Fig. 12. Building the admittance pattern for Example 1.

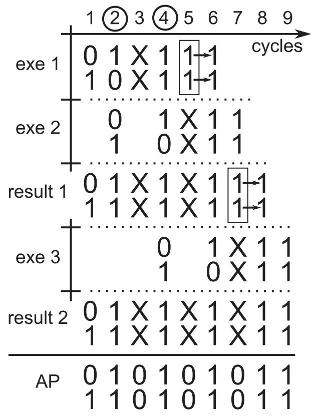

Fig. 13. Building admittance pattern for Example 2.

OR element by element of the associated columns of CP. For example, at clock cycle 3, the first execution consumption is specified by $CP_{*,3}$, the second by $CP_{*,2}$ and the third by $CP_{*,1}$. The logical OR makes the input pattern compatible with the three executions. Doing this operation for each clock cycle yields the admittance pattern. For Example 1, it produces $AP = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$.

The general process to build AP is given by Algorithm 1. For convenience, the algorithm starts by allocating and initializing an array with $L_{CP} \times n_{exe}$ as its maximum theoretical size (line 1 to 4). As explained in Example 1, we assume that the first execution starts at the clock cycle 1, thus CP is copied at the beginning of AP (line 2). For each following execution i, the algorithm searches for its triggering, which corresponds to the next insertion point of CP, named t. This is simply done by counting Δ valid columns (line 8 to 13). If at this point there are columns with only $\times$, they are skipped (line 14 to 16) because they forbid a new trigger.

Then, each column of CP must be combined with a column of AP, starting at $t' = t$ (line 17 to 36). For each couple of columns, there are four possible cases:

1. The insertion point t' is after the current end of AP, thus the column of CP is just copied (line 19 to 22).
2. The columns of CP and AP can be combined directly by a logical OR (line 23 to 25), as in Fig. 12 and Example 1. This occurs when there are no operations $\times$ OR 1.
3. The column of CP is composed of $\times$ only (line 26 to 31). If AP is not a $\times$-column, it implies to shift to the next column of AP, then copy the column of CP in the empty space.
4. The column of AP is composed of only $\times$ (line 32 to 35). The insertion point is incremented by one and no combination is needed.

Finally, the result is shrunk to its minimal size and all $\times$ are turned into 0 (line 38). This simplifies the further comparison between AP and IP. Indeed, both 0 and $\times$ represent an invalid data in AP so it is simpler to use 0 as in IP.

Algorithm 1 and these four cases are illustrated by Example 2 and Fig. 13.

Example 2. $CP = \begin{bmatrix} 0 & 1 & \times & 1 & 1 \\ 1 & 0 & \times & 1 & 1 \end{bmatrix}, P_I = 2, L_{CP} = 5, V_{CP} = 4, \Delta = 1$, and $n_{exe} = 3$.

The first row in Fig. 13 is the initial copy of CP that corresponds to the first execution. Then, the algorithm enters into the main loop (line 7) and searches (line 9 to 16) for the triggering of the second execution ($i = 2$), which is at $t = 2$. The new insertion point t' has been found and thus, CP can be combined with current AP (line 18 to 36) with the four possibilities mentioned above.

At $t' = 2$, case 2 applies: $CP_{*,1}$ and $AP_{*,2}$ can be directly combined with a logical OR. At $t' = 3$, case 4 occurs. This leads to search for a possible insertion of $CP_{*,2}$ in the next indexes, and it is effectively possible at $t' = 4$, where case 2 applies. Case 3 occurs at $t' = 5$. Since $AP_{*,5}$ is a valid column, AP is shifted right from t' (the next column figured by the small arrows), leaving an empty space where $CP_{*,3}$ can be copied. Case 2 occurs once again at $t' = 6$ and finally, case 1 occurs at $t' = 7$. At the end of the j-loop, columns 1 to 7 of AP have been modified, as shown in the row of Fig. 13 entitled result 1. The same principles are applied for the third execution that starts at $t = 4$. It yields the row entitled result 2. The bottom row shows the resulting AP after $\times$ transformations.

By construction, AP is unique and contains a cyclic part repeated a number of times depending on n_{exe}. Lines 9 to 16 determine an increment of t that may be different for several iterations of i. Since the latency between two executions is fixed and driven by Δ, the increment value is cyclic. It yields a cyclic sequence in AP, except at its beginning and/or end. The length of this sequence is simply equal to the sum of the increment values over a cycle. In Example 2, the increment is always 2 and this corresponds to the length of the cyclic part of AP, which is $\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$. For the same example, $\Delta = 2$ would result in a fixed increment of 3, and a cyclic part is $\begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$.

This building process is not totally correct when there are concurrent consumptions and CP contains null columns. In this case, for a new execution, there may be several choices of column to combine CP with AP. For example, if $CP = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}$ and $\Delta = 1$, Algorithm 1 starts by initializing AP to CP. Then, the loop in lines 9 to 13 determines that the second execution starts at $t = 2$. Nevertheless, it corresponds to a null column in CP and means that the actor can receive a valid input group at that clock cycle but it is not mandatory. It could also be at $t = 3$. Consequently, there are two possible values of t to combine CP and AP but the algorithm given here explores only the first one.

More generally, such cases lead to several possible admittance patterns. If $n_{exe} = \infty$, there may even be an infinite number of admittance patterns if such choices occur repeatedly. Nevertheless, sources are assumed to produce cyclic output patterns so each actor also receives a more or less complex cyclic input pattern. For a complete cycle, it is possible to determine the number of executions of the actor and thus to generate all possible admittance patterns by a recursive version of Algorithm 1 that explore all combinations. For concision, this version is not presented here. Moreover, it has nearly no impact on the following.

Algorithm 1 Admittance generation.

```
1  for $i = 1$ to $L_{CP} \times n_{exe}$ do
2      if $i \le L_{CP}$ then $AP_{*,i} \leftarrow CP_{*,i}$ ;
3      else $AP_{*,i} \leftarrow$ null column ;
4  end
5  $t \leftarrow 1$
6  $L_{AP} \leftarrow L_{CP}$
7  for $i = 2$ to $n_{exe}$ do
       // search start of $i^{th}$ exec.
8      $count \leftarrow 0$
9      while $count < \Delta$ do
10         if $AP_{*,index}$ is a valid column then
11             $count \leftarrow count + 1$
12         $t \leftarrow t + 1$
13     end
14     while $AP_{*,index}$ is an $\times$ column do
15         $t \leftarrow t + 1$
16     end
       // combine $CP$ with $AP$, from $t$
17     $t' \leftarrow t$
18     for $j = 1$ to $L_{CP}$ do
19         if $t' > L_{AP}$ then
20             copy $CP_{*,j}$ at $AP_{*,t'}$
21             $L_{AP} \leftarrow L_{AP} + 1$
22             $t' \leftarrow t' + 1$
23         else if $AP_{*,t'}$ and $CP_{*,j}$ can combine then
24             $AP_{*,t'} \leftarrow AP_{*,t'}$ OR $CP_{*,j}$
25             $t' \leftarrow t' + 1$
26         else if $CP_{*,j}$ is an $\times$ column then
27             if $AP_{*,t'}$ is a valid column then
28                 shift to the next column of $AP$ from $t'$
29                 $L_{AP} \leftarrow L_{AP} + 1$
30             copy $CP_{*,j}$ at $AP_{*,t'}$
31             $t' \leftarrow t' + 1$
32         else if $AP_{*,t'}$ is an $\times$ column then
33             $j \leftarrow j - 1$
34             $t' \leftarrow t' + 1$
35         end
36     end
37 end
38 remove columns of $AP$ after $L_{AP}$, turn all $\times$ into 0.
```

3.2.3. Compatibility check

When the patterns *AP* are available (at least one), we can check whether *IP* is compatible with each of them using Algorithm 2. The process is similar to the one that removes null columns of *IP* described at the beginning of this section, but here, *CP* is replaced by *AP*. In practice, there are usually some invalid data groups in the front of inputs. Thus, the algorithm firstly searches for the first valid column (line 2). Then, if *IP* is the same as *AP* or if it just contains additional null columns between two valid columns compared with *AP*, they are compatible (returning *true*). Otherwise, they are incompatible (returning *false*).

For example, Algorithm 2 applied to Example 2 with $IP = \begin{bmatrix} 0&0&0&0&0&1&0&1&0&0&1&0&1&0&1 \\ 0&0&1&0&0&1&0&1&0&0&1&0&1&0&1 \end{bmatrix}$ returns that it is compatible with *AP*. Indeed, it is easy to notice that if columns 1, 2, 4, 5 and 10 are removed, *IP* becomes equal to *AP* (with $\times$ turned into 0).

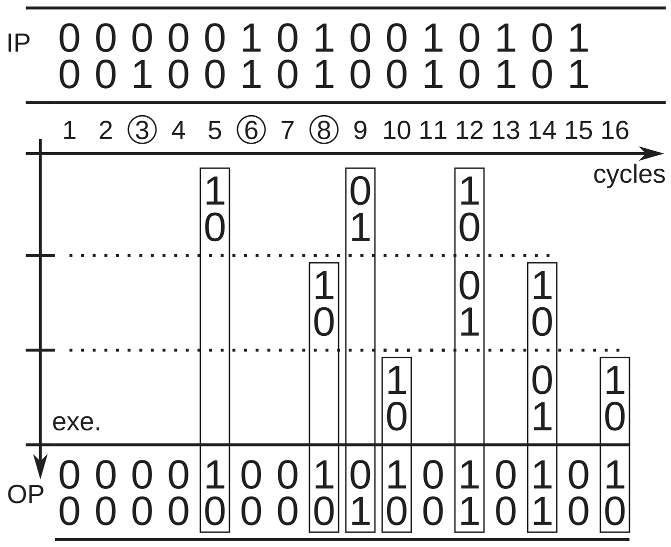

Fig. 14. The process of output pattern computation.

3.2.4. Output pattern generation

If *IP* is compatible with *AP*, *OP* can be computed using Algorithm 3.

For each execution *i*, it searches the triggering clock cycle *t* (line 6). Then, it builds a vector *II* that contains the clock cycles of the next L_{CP} groups that must be consumed during *i* relatively to *t* (line 7). After that, it loops over the valid columns in *PP* (line 8 to 14) determining the clock cycle at which they are produced. The idea is to determine the gap between the normal behavior given by *CP* and the real one given by *IP* (line 10 and 11). For example, if it needs to consume two consecutive input data to produce one output and if there is in fact an idle clock cycle between these data, the output will be delayed by one clock cycle. Thanks to the gap, *OP* is updated by combining its actual value with the current valid column in *PP*. Then, the current clock cycle *t* is incremented until the next execution. Finally, Δ valid columns are skipped (line 15).

An illustration is given by Fig. 14, completing Example 2 with $PP = \begin{bmatrix} 0&0&1&0&0&1 \\ 0&0&0&0&1&0 \end{bmatrix}$, $L_{PP} = 3$ and $PC = \begin{bmatrix} 1 & 3 & 4 \end{bmatrix}$.

For clarity, *IP* is recalled on top of the figure and the clock cycles where an execution is triggered are surrounded by a circle. The algorithm starts by computing $CI = \begin{bmatrix} 1 & 2 & 4 & 5 \end{bmatrix}$ and $PI = \begin{bmatrix} 3 & 5 & 6 \end{bmatrix}$ according to *CP* and *PP*. Then, for each execution, groups of *PP* are reported on a different row at the clock cycle computed by the algorithm.

The first execution starts at $t = 3$. According to *IP*, the next four data groups are at $t = 3, 6, 8$ and 11. Thus, relatively to t, $II = \begin{bmatrix} 1 & 4 & 6 & 9 \end{bmatrix}$. For the first output group, $PC_1 = 1$, thus $gap = II_1 - CI_1 = 0$. It leads to update *OP* at clock cycle $t + PI_1 - 1 + gap = 3 + 3 - 1 + 0 = 5$. For the second output group, $PC_2 = 3$, thus $gap = II_3 - CI_3 = 2$ and *OP* is updated at clock cycle $3 + 5 - 1 + 2 = 9$. For the third and last output group, $PC_3 = 4$, thus $gap = II_4 - CI_4 = 4$ and *OP* is updated at clock cycle $3 + 6 - 1 + 4 = 12$.

The second execution starts at $t = 6$. According to *IP*, it gives $II = \begin{bmatrix} 1 & 3 & 6 & 8 \end{bmatrix}$. Applying the same principles leads to update *OP* at clock cycles 8, 12 and 14. Nevertheless, at $t = 12$, *OP* already contains a 1. In this case, this is not a problem since that 1 is produced on the

Algorithm 2 Checking compatibility.

```
1  $t \leftarrow 1; i \leftarrow 1$                /* indexes in $IP$ & $AP$ */
2  while $IP_{*,t}$ contains only 0 do $t \leftarrow t+1$ ;
3  while $t \leq length(IP)$ do
4      if $IP_{*,t}$ != $AP_{*,i}$ then
5          if $AP_{*,i}$ contains some 1 then
6              while $t \leq length(IP)$ and $IP_{*,t}$ contains only
                0 do        /* search next group */
7                  $t \leftarrow t+1$
8              end
9              if $IP_{*,t}$ != $AP_{*,i}$ then
10                 return $false$     /* incompatible */
11             else /* compatible,$IP$ and $AP$ go one
                column further */
12                 $t \leftarrow t+1; i \leftarrow i+1$
13             end
14         else                       /* incompatible */
15             return $false$
16         end
17     else       /* compatible, $IP$ and $AP$ go one
        column further */
18         $t \leftarrow t+1; i \leftarrow i+1$
19     end
20 end
21 return $true$
```

Algorithm 3 Output pattern generation.

```
1  $CI \leftarrow$ index of valid columns in $CP$
2  $PI \leftarrow$ index of valid columns in $PP$
3  alloc. & init. $OP$ with $length(IP+PP)$ null columns.
4  $t \leftarrow 1$                          /* index in $IP$ */
5  for $i = 1$ to $n_{exe}$ do
6      while $IP_{*,t}$ is a null column do $t \leftarrow t+1$ ;
7      $II \leftarrow$ index of next $L_{CP}$ valid col. in $IP$ relatively to $t$
8      for $n = 1$ to $L_{PP}$ do
9          $p \leftarrow PI_n$
10         $c \leftarrow PC_n$
11         $gap \leftarrow II_c - CI_c$
12         $OP_{*,t+p-1+gap} \leftarrow OP_{*,t+p-1+gap}$ OR $PP_{*,p}$
13         $p \leftarrow p+1$
14     end
15     $t \leftarrow t + II_{\Delta} + 1$
16 end
17 remove null columns of $OP$ after $t-1$.
```

first output, while the 1 of the second execution is produced on the second output. Thus, updating *OP* is actually done with a logical OR. As said in Section 3.1.5, producing two 1 on the same output port and at the same clock cycle is physically impossible or in other words that the patterns are inconsistent. Fortunately, a real core correctly modeled cannot lead to such a situation. This is why Algorithm 3 does not check inconsistent cases.

3.2.5. Whole graph analysis

We consider a graph of N actors each labeled with a unique identifier $id \in \{1, N\}$. Since the graph is acyclic, an algorithm of topological sort can be used to find a traversal order $O = [O_i], i \in \{1, \ldots, N\}$, where O_i is the identifier of the ith actor to evaluate. Following this order guaranties that all predecessors of any actor are evaluated before the actor itself. It is a necessary condition for checking the compatibility. Indeed, for a given actor, the compatibility check relies on the availability of *IP*. Since it is an aggregation of *OP* (or a part of) of its predecessors, it is mandatory to evaluate them before.

The whole procedure is summarized in Algorithm 4. For the *i*th actor in the traversal order (except the sources), it consists in computing *IP*, determining its number of execution to build *AP*, checking the compatibility, and finally computing *OP*. As soon as an incompatible actor is detected, modification techniques presented in the following section are used to enforce the compatibility. Then, the analysis is resumed until the whole graph has been processed.

Algorithm 4 Analyzing the whole graph.

```
1  $O \leftarrow$ traversal order
2  for $i = 1$ to $N$ do
3      if $O_i$ is a source actor then
4          compute its $OP$
5      else
6          determine $IP$ from $OP$ of the predecessors
7          determine $n_{exe}$ from $IP$
8          compute $AP$
9          if $IP$ incompatible with $AP$ then
10             determine modifications on inputs of $O_i$
11         compute $OP$
12     end
13 end
```

4. Principles of graph modification

As shown above, ASAP solves some of the limitations of SDF-AP model discussed in Section 2.2. Nevertheless, some very simple designs may lead to incompatible patterns. In this section we present such a case and a general sketch of the developed solutions. To summarize, ASAP follows a novel approach that operates at the stream structure level contrarily to SDF models that are focused on finding a schedule and using buffers to enforce processing correctness. Since patterns reflect this structure, they can be used to find the sources of incompatibilities. Thus, it is possible to compute modifications to apply to chosen stream structures to enforce the compatibility. Since the stream structure is assumed to be cyclic, these modifications are also cyclic and can be easily translated into a VHDL process based on a state machine.

The purpose of the design given in Fig. 15 is to process a stereo signal with different filters (clipper, average and compressor) on left and right channels, before joining them to produce a mono signal. For each actor, *CP*, *PP*, *PC*, and Δ are given. These characteristics are perfectly plausible for the clipper and the average filter but have been chosen for the sake of illustration for the compressor. We investigate two configurations for the source *S*.

Firstly, we assume that *S* produces a data at each clock cycle, i.e. $PP^S = [1]$. In this case, *IP* and *CP* of the compressor are incompatible and it can be detected without the compatibility check. Indeed, if we compare what is produced by *S* and what is consumed by the compressor over a finite number of clock cycles, it is obvious that *S* produces more than the compressor consumes. We call such a situation a **sample rate inconsistency**. It can be noticed that it is not the case for the clipper and the average filter but the join is also inconsistent. Moreover, it is not possible to use a buffer to solve the problem because it would grow infinitely. A possible solution is to decimate some data streams so that all sample rates are consistent.

From a matrix Γ called topology matrix that describes the relationship between actors and what they produce/consume on their channels, we are able to check if the sample rates are consistent for each actor. If it is not the case, an algorithm crosses the graph to find the downsampling rate that must be applied on each channel to enforce the consistency. It yields a **downsampling matrix** that is used further in the process of modification. For the first configuration example, this algorithm determines a downsampling rate of $\frac{2}{3}$ for the streams before the compressor and between the average filter and the join.

The next step is to determine what kind of modification must be applied on input ports in case of incompatibility. In its present state, ASAP only considers two possibilities: delays and decimations. This is sufficient for a large number of incompatibility cases. The principle is the following. For each 1 in *IP* of an incompatible actor, a second algorithm determines if it must be delayed, decimated or kept as it is. A decimation is decided according to the donwsampling matrix and a delay to the fact that it comes before its expected place given by *AP*.

For the first configuration, this algorithm determines that the last data out of three must be decimated before the compressor, and the second out of three after the average filter. It yields:
$OP^{com.} = \begin{bmatrix}0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & \dots\end{bmatrix}$ $OP^{avg.} = \begin{bmatrix}0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & \dots\end{bmatrix}$.

Then, for the join, the algorithm determines that every 1 produced by the compressor must be delayed by two clock cycles.

The second configuration assumes that the source *S* produces a data every two clock cycles, i.e. $PP^S = \begin{bmatrix}1 & \times\end{bmatrix}$. In that case, the sample rates are consistent. Nevertheless, the join actor is still incompatible because some 1 are misplaced in its input pattern. Indeed, we have:
$OP^{com.} = \begin{bmatrix}0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & \dots\end{bmatrix}$ $OP^{clip.} = \begin{bmatrix}0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & \dots\end{bmatrix}$ $OP^{avg.} =$
$[0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 1\ \ 1\ 0\ 0 \rightarrow 1\ \ 0\ 1\ 1\ 0\ 0 \dots]$

In this configuration, the second algorithm determines that for each sequence of two 1 in $OP^{com.}$, the first one must be delayed by two clock cycles and the second one by three.

In the most simple case, delays are constant for all valid input which is very simple to translate into VHDL. Such a case occurs in the experiments presented in the following section. In other cases, since input patterns are cyclic, the decimations and delays are also cyclic. They can be represented by a state machine that uses counters on valid inputs to decide a change of state and to initiate a decimation or a delay. Thus, it is also quite easy to generate automatically their VHDL code.

5. Experiments

In order to prove the efficiency of our framework compared to the SDF-AP approach, we conducted experiments on a real application based on image processing. The design has been created with BlAsT, which is briefly described in the following section. It contains parallel processing branches and operates with two different clock domains. It is sufficiently complex to illustrate the ASAP principles and their integration in BlAsT. Moreover, even if there are few actors with relatively simple patterns, it demonstrates the huge gain in terms of resource consumption on a real architecture.

*5.1. **Bl**ock **As**sembly **T**ool*

BlAsT proposes a graphical interface to create designs. It is quite similar in its form to Simulink because it relies on panels where blocks chosen from a library can be laid and connected. The main panel represents the top group of the design and it is possible to create subgroups. Fig. 16 illustrates the graphical interface for a dummy design with two groups.

The comparison stops here because BlAsT integrates all analysis principles of ASAP presented above. It is able to test the compatibility of all blocks and to compute the modifications to apply on faulty inputs. It is also able to generate the VHDL code for the whole design and the

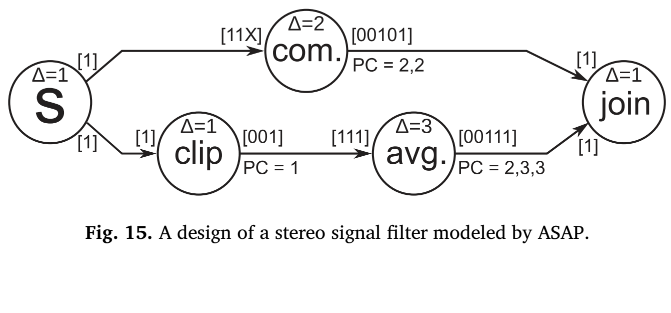

Fig. 15. A design of a stereo signal filter modeled by ASAP.

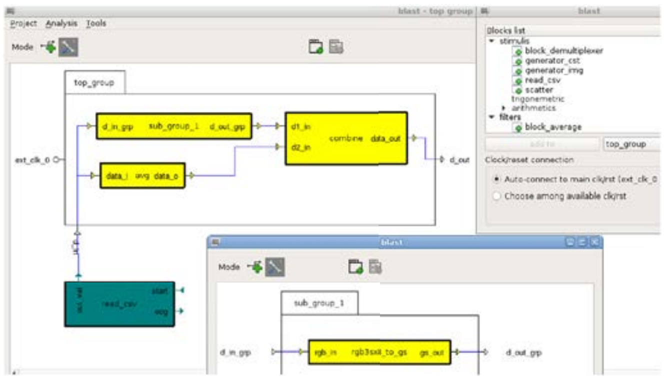

Fig. 16. The graphical interface of BlAsT.

benchmark files for its simulation. Moreover, this code is really synthesizable and can be placed and routed on a target FPGA, as demonstrated in the following section.

5.2. Example case

In this section, a realistic design is tested to show the resource saving brought by the ASAP model. It is worth noting that this design does not require complex processings and actors have quite simple patterns. This is a deliberate choice to emphasize that even a simple design may be unfeasible on a given FPGA when its implementation is based on SDF-AP and buffers, while there are no problems with an implementation based on ASAP. However, this design contains parallel branches of processing that are not synchronized, which is yet more complicated than the linear processing chains used in related works. This characteristic is used to point out the ability of ASAP to enforce a correct processing through graph modification.

Experiments are carried out on an FPGA for both ASAP model and SDF-AP model. The consumption and latency are compared. The presented case was originally implemented on a Raspberry Pi board. For the occasion, it has been adapted to the APF27 + SP Vision development board from the Armadeus company. The board hosts an iMX processor, physically linked to a Spartan 3, which can be used as it is, and also as a bridge to a Spartan 6 (in LX100 version) hosted on the SP Vision extension board. Both FPGAs are fed with an external signal at 100 MHz used to clock the design. Several jumper banks are linked to the FPGA *IO* pins of the FPGAs, allowing to bind peripherals.

The project is based on robot cars equipped with CMOS cameras. The video frames are communicated to a computation unit that detects the wheels of other cars by identifying their colored elliptic shapes. The pattern recognition starts with image manipulations before application of a Canny filter and an ellipse detector. The proof of concept presented in this section is only based on these primary manipulations because it is sufficient to clearly exhibit the advantages of our model. Fig. 17 describes this first phase, slightly modified to take the constraints of an FPGA processing into account. Its goal is to provide exploitable data to the Canny filter by converting frames into grayscale and keeping only the pixels corresponding to the wheels' color.

This process starts with a camera controller that grabs the frames from the camera. It outputs pixels in RGB24 format as a sequence of three 8 bit values (one for each component). In order to explore any desired camera configuration, we have implemented a fake version of this controller that generates a camera clock and frames of any size and any rate. Since the camera clock may differ from the external clock, the controller is followed by a FIFO to change the clock domain when necessary which enforces the rest parts of the design to work at 100 MHz. After the FIFO, pixels are converted in parallel into grayscale and YCbCr format. The latter is used as the first step to select pixels in a range of colors and luminosities. After the conversion, a deserializer outputs the three components in parallel. Each component is sent to a block that just tests if the input is greater or lesser than a parametric value and outputs true/false depending on the result of the test. The boolean values are combined by a logical AND. A threshold receives the grayscale pixels and keeps their values as they are or sets them to 0, depending on the logical AND result. Finally, a blur filter with a 3 × 3 mask is applied to prevent an incorrect behavior of the Canny filter.

The design in Fig. 17 has been created and analyzed with BlAsT according to the principles of ASAP. Whatever may be the camera clock, the analysis detects an incompatibility for the threshold block. As shown in the video demonstration available at [27], the valid inputs on its `data_in` interface are in advance compared to those on `keep_in` interface. The principles exposed in Section 4 are applied by BlAsT and yield a delay of 6 clock cycles to add on `data_in`. After this modification, the threshold block inputs are synchronized. Finally, the VHDL code is generated automatically by BlAsT. A benchmark file is also generated thanks to a block that figures the camera outside the top group. This block allows to read an image converted in a csv format and to feed the top group with a new data at each clock cycle of the camera clock. A makefile allows to compile all the sources and to launch a simulation under ISim. The whole procedure is presented in the video.

Starting from the sources generated by BlAsT, it is very easy to produce the version for SDF-AP. It merely consists in removing the validity signals and all the tests associated to them. The same principles apply for other blocks. Nevertheless, blocks that operate on a sequence of data (format conversions and blur filter) must also have a control input that indicates the beginning of the sequence. This is why there are no significant differences in terms of resource consumption between the two versions when synthesizing a single block.

Fig. 17 gives *CP* and *PP* for the ASAP version of the design. The numbers *W* and *H* are the width and height of the frames. It is worth noting that the consumption pattern of the SDF-AP version is equal to *CP* right padded with 0 until the length of *PP*. Moreover, the production pattern of the SDF-AP version is strictly equal to *PP*. This is because both

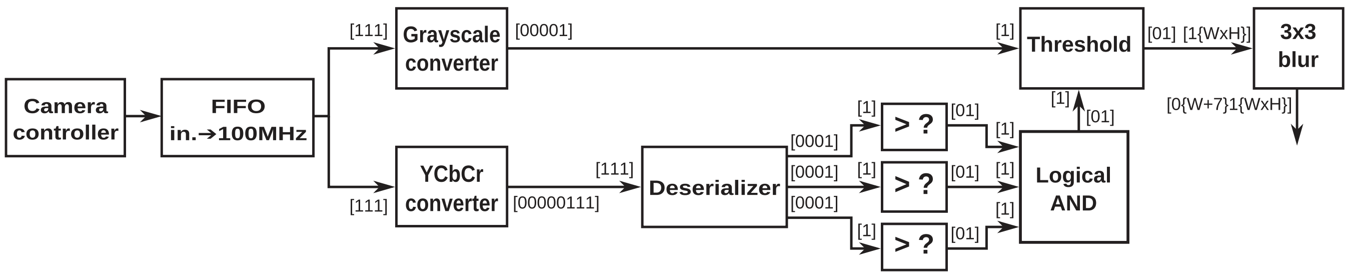

Fig. 17. Demonstration case: a graph of blocks for real-time image processing on an FPGA.

Table 1
Production counters of blocks.

Blocks	*PC*
Grayscale conv.	3
YCbCr conv.	3, 3, 3
Deserializer	3
Checker	1
AND	1
Threshold	1
Blur filter	$W+2, W+3, \ldots, W \times H-1, (W \times H)\{W + 2\}$

Table 2
Production patterns for different camera clocks.

Camera clocks	*PP*s of FIFOs
50 MHz	$(10)\{W \times H \times 3\}$
66 MHz	$(101)\{W \times H \times 3/2\}$
75 MHz	$(1011)\{W \times H\}$
80 MHz	$(10111)\{W \times H \times 3/4\}$
100 MHz	$1\{W \times H \times 3\}$

Table 3
Resources consumption with stretchable patterns.

ASAP version	
FPGA resources	Any camera clock/image size
Slice registers	283 out of 126576 (<1%)
Slice LUTs	296 out of 63288 (<1%)
8Kbits RAM blocks	4 out of 536 (<1%)
Best achievable clock period	6.886ns

Table 4
Min. and max. combination of test parameters for SDF-AP version.

SDF-AP version		
FPGA resources	128×128, 100 MHz	512×512, 66 MHz
Slice registers	390/126576 (<1%)	577/126576 (<1%)
Slice LUTs	558/63288 (<1%)	2414/63288 (4%)
8Kbits RAM blocks	16/536 (3%)	432/536 (80%)
Best achiev. clock per.	7.66ns	9.85ns

versions are based on the same code and have the same behavior when the input pattern correspond to the consumption pattern. However, if it is not the case, the SDF-AP version is not able to produce correct results, implying the use of buffers, as shown in the following.

Table 1 gives the production counters of different actors for the ASAP version. They can be deduced directly from their behavior and/or simulations. For example, the deserializer takes three serial inputs to produce three parallel outputs. Thus, it is logical that $PC = 3$. For the blur filter, outputs starts as soon as $W+2$ inputs have been consumed. This is why *PC* starts with $W+2$. But it also implies that when $W \times H$ pixels have been consumed, the filter must still produce $W+2$ outputs. This is reflected by the end of *PC* that is equal to $(W \times H)\{W + 2\}$.

Tests have been conducted using the following frame sizes: 128×128, 256×256, 512×512 and 1024×1024. We also considered five different camera clocks with five different *PP*s for the FIFO (for a single frame of size $W \times H$) summarized in Table 2. Taking into account that the camera controller produces sequentially the three pixel components (R,G and B), the camera clock corresponds to the component rate and not the pixel rate. For example, at 50 MHz, a new component is produced every 20 ns, so the FIFO outputs a component every two clock cycles. This gives a *PP* equal to 10, repeated $W \times H \times 3$ times for a whole frame. At 75 MHz, the FIFO outputs three components every four clock cycles (with the second idle). Since the output pattern of the FIFO is directly linked to the ratio between the camera clock and the external clock, the camera controller is not taken into account in the following experiments and the FIFO is considered to be the true source actor.

The most important result is that the version with stretchable access patterns does not need any extra FIFO whatever the camera rate and frame size are. Nevertheless Algorithm 2 reports an incompatibility for the threshold block because its two inputs are not synchronous. This problem is solved by inserting a simple delay of 6 clock cycles after the grayscale converter, which is totally negligible in term of resources consumption. After this light modification, all the blocks become compatible with their input streams. After synthesis and routing the design with Xilinx ISE, we obtained the results of the ASAP version shown in Table 3.

In fact, three RAM blocks are used by the blur filter to store image rows and one for the FIFO to make the clock domain conversion.

Results are totally different for the SDF-AP version. Indeed, for all camera rates except 100 MHz, a FIFO is needed before the image conversions to ensure that pixel components are streamed in three consecutive clock cycles, which is not the case just after the clock domain conversion. Furthermore, another FIFO is needed before the blur filter because the grayscale conversion leads to a pixel every three cycles.

The minimum size of these two FIFOs is directly linked to the image size (in bytes) and camera rate. For example, assuming a camera rate at 75 MHz within four clock cycles, there is an idle cycle without valid data, which has to be “removed”. With an image size of $1024 \times 1024 \times 3$, it requires a FIFO of size $(1024 \times 1024 \times 3)/4 = 786432$ bytes. Since the Spartan 6 has only 536 RAM blocks of 1 Kb, it means that it is impossible to process such an image. Moreover, since the second FIFO (before blur filter) is needed even with a camera rate at 100 MHz, the same type of computation leads to a need of 684 RAM blocks, which is once again too many.

Table 4 summarizes the metrics given by ISE for the minimal and maximal combination of test parameters. Even in the minimal configuration, SDF-AP version consumes more resources than the ASAP version. Moreover, maximum configuration leads to a very stressed design with a lot of RAM blocks used and the maximum clock path (9.85 ns) very close to the clock period (10 ns). Thus, a complete design including the Canny filter and ellipse detector could easily reach this limit.

In terms of latency, the two versions are equivalent within a few clock cycles, which is totally consistent. Even if FIFOs greatly delay the pixels, they also allow the format converters and blur filter to consume them at each clock cycle. In the ASAP version, there are no FIFO, but the blocks consume pixels slower when they are available. Thus, in both cases, the last filtered pixels are produced at nearly the same clock cycle. Taking the behavior of the blur filter into account, the global latency is roughly equal to $(W \times H \times 3 + W) \times (100/camera\_rate)$. Table 5 gives two examples of timings in elapsed clock cycles to process an image (as reported by a simulation in ISim).

These results are consistent with the above remarks. The gap between the two versions is constant and very small (5 cycles for

Table 5
Test results for two examples of timings.

	128×128		512×512	
	ASAP	SDF-AP	ASAP	SDF-AP
75 MHz	65679	65687	1049103	1049111
100 MHz	49296	49301	786960	786965

100 MHz). The formula given above is also verified. For example, for a size of 512 $\times$ 512 and at 75 MHz, it gives 1049258 cycles, which is very close from the simulation result.

6. Conclusions

We have recalled the SDF-AP model that constitutes a major improvement compared with the elementary SDF model. Then, we have pointed out some of its limitations through illustrative examples regarding concurrency, strict pattern conformance and buffering. They result in a limited model of actors' behaviors and possibly infeasible designs after analysis. They also cause waste of logic resources when implementing a design on a real architecture. Then, we have introduced the concept of Actors with Stretchable Access Patterns (ASAP), a novel way to address the scheduling problem of actors. It allows to model and analyze the actors' behaviors taking their implementation on a real architecture into account. This modeling approach solves the limitations of concurrent executions and buffering problems. Some algorithms for patterns generation and checking compatibility are provided. The ease of transposition of the model into VHDL codes and the saving of logic resources are proved through experiments conducted with BlAsT, a new EDA tool that allows to create and to analyze FPGA designs with the principles of ASAP.

Future works will focus on addressing the problem of loops in the graph. Indeed, there are a lot of scientific applications based on designs with feedbacks, notably in the domain of control. The principles of graph modifications can also be improved. For example, some cases lead to increasing delays on a bounded sequence of valid inputs. It could be solved by a simple buffer with a size given by the maximum delay. Moreover, the modifications are searched at the actor level but better solutions could be found taking the whole graph into account. Finally, BlAsT is in a prototype state and should be enriched with a large library of blocks and a better integration of the existing FPGA architectures.

Acknowledgment

This work is supported by the EIPHI Graduate School (contract ANR-17-EURE-0002) and Doctoral Research Fund of Shandong Jianzhu University (XNBS1850). The authors would also like to thank the reviewers for their helpful suggestions to improve the quality of this paper.

Appendix A. Supplementary data

Supplementary data to this article can be found online at https://doi.org/10.1016/j.vlsi.2019.01.001.

References

[1] E.A. Lee, D.G. Messerschmitt, Synchronous data flow, Proc. IEEE 75 (9) (1987) 1235–1245.

[2] J. Eker, J.W. Janneck, Cal Language Report: Specification of the Cal Actor Language, Tech. rep., University of California at Berkeley, 2003.

[3] M. Wipliez, G. Roquier, J.-F. Nezan, Software code generation for the rvc-cal language, J. Signal Process. Syst.

[4] J. Serot, F. Berry, High-level dataflow programming for reconfigurable computing, in: 2014 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW), vol. 00, 2014, pp. 72–77, https://doi.org/10.1109/SBAC-PADW.2014.18, http://doi.ieeecomputersociety.org/10.1109/SBAC-PADW.2014.18.

[5] R. Townsend, M.A. Kim, S.A. Edwards, From functional programs to pipelined dataflow circuits, in: Proceedings of the 26th International Conference on Compiler Construction, CC 2017, ACM, New York, NY, USA, 2017, pp. 76–86, https://doi.org/10.1145/3033019.3033027.

[6] S.A. Edwards, R. Townsend, M.A. Kim, Compositional dataflow circuits, in: Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE '17, ACM, New York, NY, USA, 2017, pp. 175–184. http://doi.acm.org/10.1145/3127041.3127055.

[7] E.A. Lee, D.G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Trans. Comput. 100 (1) (1987) 24–35.

[8] M. Geilen, S. Tripakis, M. Wiggers, The earlier the better: a theory of timed actor interfaces, in: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, ACM, 2011, pp. 23–32.

[9] S. Stuijk, M. Geilen, T. Basten, Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs, IEEE Trans. Comput. 57 (10) (2008) 1331–1345.

[10] M. Engels, G. Bilson, R. Lauwereins, J. Peperstraete, Cycle-static dataflow: model and implementation, in: Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on, vol. 1, IEEE, 1994, pp. 503–507.

[11] G. Bilsen, M. Engels, R. Lauwereins, J.A. Peperstraete, Cyclo-static data flow, in: Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on, vol. 5, IEEE, 1995, pp. 3255–3258.

[12] A. Girault, B. Lee, E.A. Lee, Hierarchical finite state machines with multiple concurrency models, IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 18 (6) (1999) 742–760.

[13] W. Liu, M. Yuan, X. He, Z. Gu, X. Liu, Efficient sat-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization, in: Real-Time Systems Symposium, 2008, IEEE, 2008, pp. 492–504.

[14] W. Plishker, N. Sane, S.S. Bhattacharyya, A generalized scheduling approach for dynamic dataflow applications, in: Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, 2009, pp. 111–116.

[15] S. Stuijk, M. Geilen, B. Theelen, T. Basten, Scenario-aware dataflow: modeling, analysis and implementation of dynamic applications, in: Embedded Computer Systems (SAMOS), 2011 International Conference on, IEEE, 2011, pp. 404–411.

[16] S. Tripakis, H. Andrade, A. Ghosal, R. Limaye, K. Ravindran, G. Wang, G. Yang, J. Kornerup, I. Wong, Correct and non-defensive glue design using abstract models, in: Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on, IEEE, 2011, pp. 59–68.

[17] A. Ghosal, R. Limaye, K. Ravindran, S. Tripakis, A. Prasad, G. Wang, T.N. Tran, H. Andrade, Static dataflow with access patterns: semantics and analysis, in: Proceedings of the 49th Annual Design Automation Conference, ACM, 2012, pp. 656–663.

[18] M. Benazouz, O. Marchetti, A. Munier-Kordon, T. Michel, A new method for minimizing buffer sizes for cyclo-static dataflow graphs, in: Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on, IEEE, 2010, pp. 11–20.

[19] S. Stuijk, M. Geilen, T. Basten, Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs, IEEE Trans. Comput. 57 (10) (2008) 1331–1345.

[20] H. Kee, S.S. Bhattacharyya, J. Kornerup, Efficient static buffering to guarantee throughput-optimal fpga implementation of synchronous dataflow graphs, in: Embedded Computer Systems (SAMOS), 2010 International Conference on, IEEE, 2010, pp. 136–143.

[21] K. Ravindran, A. Ghosal, R. Limaye, G. Wang, G. Yang, H. Andrade, Analysis techniques for static dataflow models with access patterns, in: Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on, IEEE, 2012, pp. 1–8.

[22] G. Wang, R. Allen, H.A. Andrade, A. Sangiovanni-Vincentelli, Communication storage optimization for static dataflow with access patterns under periodic scheduling and throughput constraint, Comput. Electr. Eng. 40 (6) (2014) 1858–1873.

[23] M. Wiggers, M. Bekooij, M. Bekooij, G. Smit, Computation of buffer capacities for throughput constrained and data dependent inter-task communication, in: Design Automation and Test in Europe, No. 1, EDA Consortium, 2008, pp. 640–645, https://doi.org/10.1109/DATE.2008.4484749.

[24] S. Tripakis, R. Limaye, K. Ravindran, G. Wang, H. Andrade, A. Ghosal, Tokens vs. signals: on conformance between formal models of dataflow and hardware, J. Signal Process. Syst. 85 (1) (2016) 23–43.

[25] G.R. Gao, R. Govindarajan, P. Panangaden, Well-behaved dataflow programs for dsp computation, in: [Proceedings] ICASSP-92: 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 5, 1992, pp. 561–564, https://doi.org/10.1109/ICASSP.1992.226558.

[26] J. Boutellier, J. Wu, H. Huttunen, S. Bhattacharyya, Prune: dynamic and decidable dataflow for signal processing on heterogeneous platforms, IEEE Trans. Signal Process. PP (2017) 1, https://doi.org/10.1109/TSP.2017.2773424.

[27] A demonstration of BlAsT: Block assembly tool, http://bilbo.iut-bm.univ-fcomte.fr/staff/sdomas/blastdemo.mkv.