

## Translating Data Flow to Synchronous Block Diagrams

Roberto Lubliner\*  
The Pennsylvania State University  
University Park, PA 16802, USA  
rluble@psu.edu

Stavros Tripakis  
Cadence Research Laboratories  
Berkeley, CA 94704, USA  
tripakis@cadence.com

### Abstract

We propose a method to automatically transform synchronous data flow diagrams into synchronous block diagrams. The idea is to use triggers, a mechanism that allows a block to be fired only at selected times. We discuss how to extend the transformation to also cover dynamic data flow diagrams where the number of tokens produced and consumed by blocks is variable. Our method allows widespread tools such as Simulink which are based on the synchronous block diagram model to be used for data flow diagrams as well.

### 1 Introduction

Data flow diagrams is a widespread model for signal processing and multimedia applications. Many data flow models have been proposed in the literature, from general ones such as Kahn process networks [3], to more restricted ones such as *synchronous data flow* (SDF) [5]. The trade-offs in these models are between expressiveness (what can be modeled, how easily, etc.) and analyzability (what can be computed, how efficiently, etc.).

Tools that support the capture and simulation of such models exist, both in the academia (see, for instance, the Ptolemy project<sup>1</sup>) as well as the industry (see, for instance, Agilent's ADS<sup>2</sup>). These tools however, are not as widespread as tools such as Simulink from The MathWorks<sup>3</sup>.

Simulink supports another popular model of computation, that of synchronous block diagrams. This model is particularly useful for embedded control applications, for instance, X-by-wire. Our motivation in this paper is to understand to what extent a tool such as Simulink could be

used to capture data flow models such as the ones described above. This is beneficial not only because of the wider distribution of Simulink, but also because it supports multiple other features, not usually supported by data flow tools. For instance, Simulink allows the composition of discrete-time with continuous-time models (where discrete-time signals are viewed as piecewise-constant continuous-time signals). Simulink is also supported by code generators such as Real-Time Workshop.

Our main contribution is to propose a scheme to transform automatically SDF diagrams to synchronous block diagrams (SBDs). This scheme relies on previous techniques to analyze SDF diagrams developed in [5] and other works. These techniques produce a periodic schedule (if one exists) for a given SDF diagram, that guarantees that the execution will never block and will use a finite amount of memory. Thus, a static block diagram formalism such as SBDs can be used as a target language.

The main idea is to exploit the feature of *triggers* available in SBDs (as well as Simulink) which allows blocks to be fired at different rates. This allows us a transformation of SDF diagrams to SBDs which is *modular*, that is, where the changes to the blocks themselves are minimal and straightforward. Once this idea is used, the transformation itself is not inherently difficult, but it does involve tedious book-keeping. Automation of this task helps avoid errors that could be introduced if a manual transformation is attempted.

We also discuss how the above transformation can be extended to dynamic data flow diagrams, where the number of tokens produced and consumed is generally unknown at compile-time, and is instead determined at run-time. The principle is again to use triggers, with the difference that the triggering policy decides dynamically which blocks to trigger. The idea is to trigger blocks as soon as they can be triggered, that is, as soon as they have enough tokens at their inputs and enough free space at their outputs.

\*Part of this work was performed while the author was on an internship at Cadence Research Laboratories.

<sup>1</sup><http://ptolemy.eecs.berkeley.edu/>

<sup>2</sup>[http://eesof.tm.agilent.com/products/ads\\_main.html](http://eesof.tm.agilent.com/products/ads_main.html)

<sup>3</sup><http://www.mathworks.com/products/simulink/>

## 2 Synchronous block diagrams with triggers

In this section we describe our notation which is a simple form of block diagrams, and its semantics, which is synchronous. In its full form, the notation is *hierarchical*, meaning that diagrams can be encapsulated into *macro* (or *composite*) blocks, that can themselves be connected with other blocks and further encapsulated. For the purposes of this paper, however, we do not need this feature and can use *flat* (non-hierarchical) diagrams. We refer the reader to [7, 6] for more details on synchronous block diagrams.

A synchronous block diagram (SBD) consists of a set of blocks connected by a set of links. See Figure 1 for an example. Each block has a set of input ports and a set of output ports. Diagrams are formed by connecting the output port of a block  $A$  to the input port of a block  $B$ . An output port can be connected to more than one input ports. However, an input port can only be connected to a single output. Multiple instances of the same block are allowed in an SBD (e.g., in Figure 1, block  $A$  is instantiated twice). When we refer to a “block” we refer to a block instance.

A block  $A$  of an SBD may be *triggered* by a link  $x$ :  $x$  is then called the *trigger* of  $A$ . The intension is that  $A$  is to “fire” only when the value carried by  $x$  is true. A block can have at most one trigger. The SBD of Figure 1 has one triggered block,  $B$ .

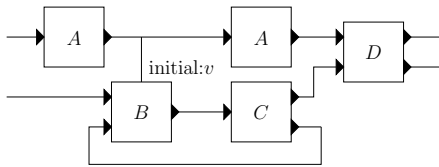


Figure 1. A block diagram.

When a block is triggered, the user specifies initial values for each output of that block. These determine the values of the outputs during the initial period (possibly empty) until the block is triggered for the first time. In the SBD of Figure 1, an initial value  $v$  is specified for the (single) output of triggered block  $B$ .

Each block  $A$  can be classified as either *combinational* (state-less) or *sequential* (having internal state). Some sequential blocks are *Moore-sequential*. Each output of a Moore-sequential block *only depends on the current state, but not on the current inputs*. For example a *unit-delay* block that stores the input and provides it as output in the next synchronous instant is a Moore-sequential block.

We assign semantics only to *acyclic* diagrams: an SBD is acyclic if every feedback loop contains at least one Moore-sequential block, from an input to an output. Notice that feedback loops with triggers are allowed, however, the

acyclicity condition requires the loop to contain a Moore-sequential block other than the triggered block. The SBD of Figure 1 is acyclic iff  $B$  or  $C$  is a Moore-sequential block (also if both are).

The semantics we use are standard synchronous semantics used also in languages like Lustre. We interpret every link  $x$  of the diagram as a *signal*, which is a total function  $x : \mathbb{N} \rightarrow V_x$ , where  $\mathbb{N} = \{0, 1, 2, 3, \dots\}$  and  $V_x$  is a set of values:  $x(k)$  denotes the value of signal  $x$  at time instant  $k$ . If  $x$  is an input this value is determined by the environment, otherwise it is determined by the (unique) block that produces  $x$ . Call this block  $A_x$ . If  $A_x$  is triggered by some signal  $t$ , then for each instant  $k$ , in order to determine  $x(k)$  we first need to determine  $t(k)$ : if  $t(k)$  is false, then  $x(k) = x(k - 1)$  (if  $k = 0$  then  $x(k) = v$  where  $v$  is the initial value specified by the user); if  $t(k)$  is true then  $x(k)$  is defined by “firing” the block that produces  $x$ , just like in the case where  $A$  is not triggered. If  $A_x$  is not triggered then the semantics is the same as if  $A_x$  were triggered by a signal  $t$  such that  $t(k)$  is true for all  $k$ . Since the diagram is acyclic there exists a well-defined order of firing the blocks to compute the values of all signals at a given synchronous instant.

It is worth noting that triggers do not add to the expressive power of SBDs and can be eliminated by a structural transformation that preserves the semantics: this transformation can be done hierarchically and atomic blocks (at the leaves of the hierarchy) can be eliminated provided they are combinational or unit-delay blocks [6]. Even though triggers can be eliminated in principle, it is useful not to do so, for modularity and other reasons: see [6] for details.

## 3 Synchronous Data Flow

Synchronous Data Flow (SDF) diagrams were introduced in [5] as a special case of general data flow diagrams, where the number of tokens produced and consumed by each “firing” of a block is statically specified. Similar models had been studied also before, e.g., see [4, 9].

An example SDF diagram is shown in Figure 2 (top). It consists of two blocks,  $A$  and  $B$ , connected through a link (for simplicity, we use similar graphical notation for SDF diagrams as for SBDs). Block  $A$  has one output port and block  $B$  has one input port. The annotations “2” and “3” on these ports specify the number of *tokens* (pieces of data) produced and consumed by the corresponding blocks, every time these blocks are fired. Thus, every time  $A$  is fired it produces 2 tokens at its unique output port (it consumes no token since it has no input ports). Every time  $B$  is fired it consumes 3 tokens from its unique input port.

Note that, although  $A$  can fire at any time,  $B$  can only fire when enough tokens are available at its input port. For instance, if initially there are no tokens in the link,  $B$  can-



**Figure 2.** Example of SDF diagram and transformation into homogeneous SDF proposed by Lee and Messerschmitt.

not fire. Even after  $A$  fires once,  $B$  still cannot fire, since  $A$  produced only 2 tokens, but 3 are required for  $B$  to fire. If  $A$  is fired twice before  $B$ , then 4 tokens are available, and  $B$  can fire. After the firing sequence  $A; A; B$ , one token remains. After firing  $A; A; B; A; B$  no tokens remain. Thus, the previous sequence can be repeated *ad infinitum*, producing an infinite stream of tokens transmitted from  $A$  to  $B$  through the link. Notice that in order to implement the above schedule, a buffer that can store 4 tokens is needed.

A periodic schedule like the one above is *admissible* in the sense that (1) node precedences are respected, that is, a node cannot execute before all required input tokens are available, (2) execution does not block, and (3) a finite amount of buffering is required to store tokens until they are consumed. For some SDF diagrams, periodic admissible schedules do not exist: some diagrams require infinite buffers to store all tokens, while some other diagrams may deadlock. [5] discusses how to check whether a given SDF diagram has a periodic admissible schedule.

A special case of SDF diagrams is the one where each block consumes a single token from each input port and produces a single token at each output port, that is, all ports are annotated by “1”. These are called *homogeneous* SDF diagrams in [5]. Homogeneous SDF diagrams are equivalent to SBDs without triggers.

An interesting question is whether a general SDF diagram can be transformed to a homogeneous SDF diagram. In [5] it is mentioned that such a transformation is possible, but it generally requires replicating some of the blocks. An example is shown in Figure 2, where the top diagram is transformed to the bottom one, which is homogeneous.

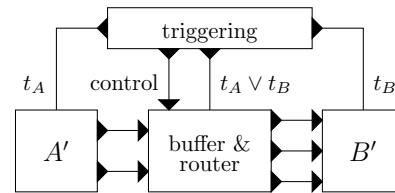
This transformation is valid only when the replicated blocks are combinational, that is, without internal state. Otherwise, replication causes the states of the replicas to be updated erroneously. This is because each replica has its own state, which is not shared with the other replicas. In what follows, we propose a different method to transform SDF diagrams to synchronous block diagrams with triggers, that avoids this problem.

## 4 From SDF diagrams to synchronous block diagrams

We propose a transformation only for SDF diagrams that have a periodic admissible schedule. Our transformation takes as input an SDF diagram and a periodic admissible schedule for this diagram. It produces as output an SBD which is equivalent to the input SDF diagram.

We should clarify at this point what we mean by “equivalent”. What we mainly want to preserve is the sequence of tokens produced in the SDF diagram. This can be seen as an “untimed” semantics of SDF. Since the notions of time in SDF and SBDs are different, we do not want at this point to impose preservation of timing constraints.

The token streams of an SDF diagram can be defined as follows. Every link of the diagram defines one stream, that is, an infinite sequence of tokens, produced by the source block of the link. For example, in Figure 2, the link between  $A$  and  $B$  defines a stream of tokens produced by  $A$ . When  $A$  is fired for the  $k$ -th time, it produces two tokens, with values  $a_k^1$  and  $a_k^2$ . Then, the stream is the sequence of pairs:  $(a_0^1, a_0^2)(a_1^1, a_1^2) \dots$ .



**Figure 3.** Transformation of the SDF diagram of Figure 2 into a synchronous block diagram.

The outline of the transformation we propose in this paper is shown in Figure 3. This figure shows the outline of the synchronous block diagram generated from the SDF of Figure 2 (top). The major points in our transformation are the following.

First, every block  $A$  that appears in the SDF is “copied” once into a corresponding block  $A'$  in the synchronous block diagram.  $A'$  is essentially identical to  $A$ , except for its interface, which changes slightly. In particular, for every output port  $p$  of  $A$ , if  $p$  produces  $n$  tokens at each firing, then  $A'$  has  $n$  output ports  $p^1, \dots, p^n$ , corresponding to the single port  $p$  of  $A$ . If  $A$  produces the tuple  $(a^1, \dots, a^n)$  at port  $p$  at a given firing, then  $A'$  must produce  $a^1$  at  $p^1$ ,  $a^2$  at  $p^2$ , and so on. We do a similar modification to the input ports.

Second, for every link in the SDF diagram, there is a “buffer & router” (B&R) block in the synchronous block diagram, that connects the corresponding source and destination blocks. One such routing block is shown in Fig-

ure 3, corresponding to the unique link between  $A$  and  $B$  in the original SDF diagram. The function of these routing blocks and how they can be generated automatically is described below. Routing blocks are generally sequential blocks, since they need memory to store the tokens produced by producer blocks, until consumer blocks fire and consume these tokens.

Finally, the synchronous block diagram contains a single “triggering” block that produces a trigger for each block in the diagram, except those blocks that will be fired at every synchronous instant (see below). In Figure 3, the triggering block produces three triggering signals:  $t_A$ , the trigger of  $A'$ ,  $t_B$ , the trigger of  $B'$ , and the trigger of the B&R block: the latter is obtained as the disjunction (logical OR) of  $t_A$  and  $t_B$ . In other words, the B&R block is triggered iff its writer block or its reader block (or both) are triggered.

The triggering block also produces a “control” signal which is fed to the B&R block as an input (notice that the control signal is an input of B&R, not a trigger). The function of the control signal will be explained below. It should be noted that the control signal is not really necessary in the SDF case, where all control information is statically available. Thus, this signal can be optimized away (see below). However, it is still interesting to use this presentation because it extends naturally to less “static” data-flow diagrams, which can also be transformed into SBDs. See discussion later in this paper.

Note there is a single triggering block for the entire diagram. The triggering block essentially implements the periodic schedule of the SDF. The triggering block is a Moore-sequential block.

#### 4.1 Generation of the triggering block

The triggering block is generated from the periodic schedule of the SDF. Suppose this schedule is a sequence of the form  $A_{i_1}; A_{i_2}; \dots; A_{i_n}$ . A simple way to implement the triggering block is to have it trigger a single block at a time:  $A_{i_1}$  at instant  $k = 0$ ,  $A_{i_2}$  at instant  $k = 1, \dots, A_{i_n}$  at instant  $k = n - 1$ , then repeat with  $A_{i_1}$  at instant  $k = n$ , etc. However, this is wasteful, since only one block is triggered at a given synchronous instant, whereas more could be triggered simultaneously.

To see this, consider our running example of Figures 2 and 3. Suppose the periodic schedule is:  $A; A; B; A; B$ . Then, the triggering policy described above would have period 5 (the length of the schedule) and trigger  $A$  at instant 0,  $A$  at instant 1,  $B$  at instant 2, and so on. We use the notation  $A; A; B; A; B$  for this triggering policy (for simplicity, we don't use primed symbols in representing the triggering policy).

An alternative triggering policy is the following, with period 3: trigger  $A$  at instant 0; trigger  $A$  and  $B$  at instant 1;

trigger  $A$  and  $B$  at instant 2; repeat. We use the notation  $A; (A, B); (A, B)$  for this triggering policy. The alternative policy may be preferred for the reason that has a shorter period and executes more blocks at the same synchronous instant. Also, as we shall see below, this policy results in smaller memory requirements.

Generating the alternative triggering policy from the SDF periodic schedule is simple. It suffices to decompose a given schedule  $\rho$  into a set of segments  $\rho_1, \dots, \rho_k$ , such that  $\rho$  is the concatenation  $\rho_1\rho_2 \dots \rho_k$ , and in every segment  $\rho_i$ , every block appears at most once. In this case, this “greedy” algorithm provides the optimal triggering policy in terms of memory requirements, however, this is not generally the case. It is beyond the scope of this paper to study the problem of optimal buffer requirements, and we refer the reader to the literature for existing work on this topic (e.g., [2]).

#### 4.2 Generation of the buffer & router blocks

Once the triggering policy is determined using one of the methods described above, the B&R blocks can be generated. Notice that the operation of the B&R blocks generally depends upon the triggering policy: for instance, in the example above, this operation will be different if the triggering with period 5 is chosen, than if the triggering with period 3 is chosen.

Consider a B&R block linking a writer block  $A'$  and a reader block  $B'$ , as in Figure 3. We distinguish two cases: (a) where the triggering policy guarantees that  $A'$  and  $B'$  are never triggered at the same time, i.e.,  $t_A \wedge t_B$  (the conjunction of signals  $t_A$  and  $t_B$ ) is never true; (b) the negation of case (a).

In case (a), the B&R block implements essentially a FIFO queue. Every time the B&R block is triggered, one (and only one) of two operations can be performed on this queue: either the writer block appends a pre-determined number of elements to the end of the queue (we call this a *put*), or the reader block removes a pre-determined number of elements from the head of the queue (we call this a *get*). The control signal received by the B&R block from the triggering block specifies what the operation to be performed is. Therefore, the domain of the control signal is binary in this case:  $\forall k \in \mathbb{N} : \text{control}(k) \in \{\text{put}, \text{get}\}$ .

In case (b), the B&R block implements something slightly more complex than a standard FIFO queue. The difference is that a put and a get can happen simultaneously in this case: this happens when both  $t_A$  and  $t_B$  are true, that is both the writer and the reader blocks are triggered at the same time. We call this operation a *put-get* (a simultaneous put and get). The semantics of put-get is equivalent to a put followed by a get, however, the buffer requirements for its implementation are different, as explained below. In this case, the domain of the control signal received by the

B&R block from the triggering block is ternary:  $\forall k \in \mathbb{N} : \text{control}(k) \in \{\text{put}, \text{get}, \text{put-get}\}$ .

Let us use our running example of Figure 3 to illustrate the above notions. First, consider the triggering policy having period 5:  $A; A; B; A; B$ . This policy falls in case (a). In this case, it can be seen that the FIFO queue needs to have size 4, to store the tokens produced by the first two invocations of  $A$  at each period.

Next, consider the triggering policy of period 3:  $A; (A, B); (A, B)$ . This policy falls in case (b). In this case, the size of the queue need not be more than 2. Indeed, at time  $k = 0$ ,  $A$  fires and produces two tokens,  $(a_0^1, a_0^2)$ , that are stored in the queue. At time  $k = 1$ , both  $A$  and  $B$  fire.  $A$  produces two new tokens,  $(a_1^1, a_1^2)$ . But since  $B$  fires, it consumes the first three tokens produced by  $A$ , namely,  $a_0^1, a_0^2, a_1^1$ . Therefore, at the end of the instant  $k = 1$ , only one token needs to be stored in the queue, namely  $a_1^2$ . At  $k = 2$ ,  $A$  produces two new tokens,  $(a_2^1, a_2^2)$ . But since  $B$  fires, it consumes the last three tokens produced by  $A$ , namely,  $a_1^2, a_2^1, a_2^2$ . Therefore, at the end of the instant  $k = 2$ , the queue is empty, and the operation can repeat itself.

We proceed to discuss implementation of the B&R blocks, in each of the cases (a) and (b) listed above. One B&R block has to be generated for each link in the diagram. The size of the FIFO queues of each of the B&R blocks needs to be determined first. This is easy to do, once the triggering policy has been fixed. Recall that the latter is computed from the periodic schedule of the SDF, and note that the size of the queues generally depends not only on the triggering policy, but also on the periodic schedule itself. For instance, in our running example, if the periodic schedule is fixed to  $A; A; A; B; B$ , then the queue size cannot be smaller than 4: this is achieved with the triggering policy  $A; A; (A, B); B$ . On the other hand, we have seen that, given the schedule  $A; A; B; A; B$ , we can achieve a better queue size, namely 2. How to generate memory-optimal periodic schedules is beyond the scope of this paper. This problem has been extensively studied in the literature, see for instance [2].

To compute the size of queues, given a triggering policy  $P$ , proceed as follows. Let  $P = P_1; P_2; \dots; P_n$ , where  $P_i$  is a set of blocks (each  $P_i$  is a partially ordered set). For each queue  $Q$ , assign a sequence  $c_Q^0, c_Q^1, \dots, c_Q^n$  of integers, with  $c_Q^0 = 0$ . Let  $A$  be the writer block of  $Q$  and  $B$  the reader of  $Q$  and let  $w_A$  be the number of tokens produced by  $A$  to  $Q$ , and  $r_B$  the number of tokens consumed by  $B$  from  $Q$ . Define, for  $i = 1, \dots, n$ :  $c_Q^i = c_Q^{i-1} + w_A^i - r_B^i$ , where: if  $A \in P_i$  then  $w_A^i = w_A$ , else  $w_A^i = 0$ ; if  $B \in P_i$  then  $r_B^i = r_B$ , else  $r_B^i = 0$ . Then, the size of  $Q$  needs to be greater or equal to  $\max\{c_Q^1, \dots, c_Q^n\}$ . Notice that, by definition of the validity of the periodic schedule (and thus also of the triggering policy) none of the values  $c_Q^i$  can be

```
BR_n_m( inputs p[n], control; outputs, q[m] ) {
  local FIFO queue Q;
  step(){
    ctrl := control.read();
    if (ctrl = 'put')
      for (i = 1 to n) Q.put( p[i].read() )
    else if (ctrl = 'get')
      for (j = 1 to m) q[j].write( Q.get() )
    else /* ctrl = 'put-get' */
      i := 1;
      for (j = 1 to m)
        if (Q.no_elems() > 0)
          q[j].write( Q.get() );
        else
          q[j].write( p[i++].read() )
      while (i <= n) Q.put( p[i++].read() )
  }
}
```

Figure 4. Generic implementation of a B&R block.

negative.

Once the size of the queues is determined, the B&R block can be implemented as follows. In case (a), the B&R block can be implemented as a circular buffer. Moreover, the implementation of this buffer can be simplified due to the fact that it is guaranteed that a put operation will never overflow and a get operation will never underflow the queue (this is guaranteed by the validity of the schedule). An additional function of B&R blocks is “routing”. This can also be implemented in standard ways, using appropriate multiplexers and a sufficiently dense connectivity mesh.

Let us illustrate the implementation of a B&R block in case (b), which is more interesting. A *behavioral* description of such a block is shown in Figure 4. The block links a writer which produces  $n$  tokens to a reader which consumes  $m$  tokens. Thus, the block has an array of input ports of size  $n$  and an array of output ports of size  $m$ . The block also has an input port for the control signal.

The block uses a local FIFO queue  $Q$ , which could be implemented as a circular buffer, as mentioned above.  $Q$  provides the services  $Q.put(e)$  (put element  $e$ ),  $Q.get()$  (get and return first element), and  $Q.no_elems()$  which returns the number of elements currently in the queue.

When the block is triggered, its  $step()$  function is called. Depending on the operation to be performed, this function does the following: If the operation is a ‘put’, all input ports are read, and the results are stored in the queue. If the operation is a ‘get’, the first  $m$  elements are removed from the queue and written to the output ports. If the operation is a ‘put-get’, then the following steps are performed: first, the queue is emptied and the stored elements are written in the output ports; then, if there are still tokens remaining to be written in the outputs, this is done by reading directly from the input ports (“feed-through”); finally, the remaining inputs, if any, are read and stored in the queue. Notice that this implementation assumes that the caller guarantees absence of overflow or underflow, which is true in our case.

Such a behavioral description can be easily transformed

into a *structural* description, that is, an SBD. For instance, *logic synthesis* methods similar as those used for hardware synthesis can be used here as well.

It should be noted that the implementation described above is quite generic, in the sense that it admits any sequence of ‘put’, ‘get’, or ‘put-get’ operations, as long as this sequence avoids overflow or underflow. In the case of the transformation of SDF diagrams into SBDs, we have more information available, namely, we know exactly the sequence of operations that is going to be applied: this is specified in the triggering policy. Therefore, we can use this information to specialize (and potentially optimize) the above generic implementation.

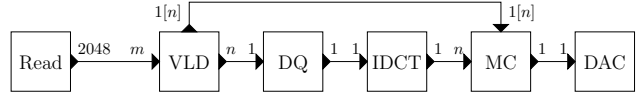
For instance, consider our running example of Figures 2 and 3, and the triggering policy  $A; (A, B); (A, B)$ . In this case, the (repeated) sequence of operations is ‘put’, ‘put-get’, ‘put-get’. Therefore, we can eliminate the control signal from the B&R block and maintain a counter modulo 3, which keeps track of the current operation to be performed. The counter is incremented each time the B&R block is triggered. We omit the details of this implementation, as it can be derived from the generic one in a straightforward manner.

### 5 Transforming dynamic data flow diagrams

We have presented a method to transform synchronous data flow diagrams to synchronous block diagrams. SDF diagrams are *static* in the sense that the number of tokens produced and consumed by the blocks is constant and known at compile-time. However, in many applications of interest, this number is not constant, and generally depends on the execution and the input data. An example of such an application is the H.263 video decoder presented in [10]. That paper also proposes a *dynamic* data flow model that allows to capture this kind of applications, as well as an algorithm to compute buffer capacities in order to guarantee a certain throughput.

In this section, we discuss how our transformation method can also be applied to dynamic data flow diagrams. We use the H.263 example of [10], which is reproduced in Figure 5. The Reader reads blocks of 2048 bytes from a source (e.g., compact disk). The number of bytes  $m$  consumed by the VLD (variable-length decoder) block and the number of blocks  $n$  produced by it are both data-dependent. VLD communicates the value  $n$  to the MC (motion-compensator) block so that the latter can assemble the picture.

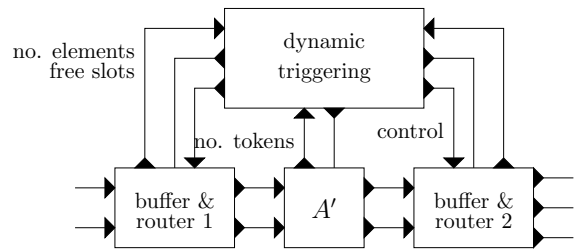
The principle of transforming such a diagram to an SBD is the same as the transformation from SDF diagrams we proposed above: we use triggers to fire blocks at selective times. However, in the case of dynamic data flow diagrams, there is not static periodic schedule available. Thus, we do



**Figure 5.** A model of the H.263 video decoder as a dynamic data flow diagram (from Wiggers et al).

not know *a priori* when a block needs to be fired. To overcome this problem we will use the following simple idea: fire a block as soon as it can be fired, that is, as soon as there are enough available tokens in its inputs and enough free space in its outputs.

Let us make this more precise. In order for the triggering block to know whether a block  $A$  can be fired, it needs the following information: (1) how many tokens  $A$  needs to consume from each of its input ports (notice that these numbers are variable); (2) how many tokens are currently available on each of the input B&R blocks of  $A$ ; (3) how many tokens  $A$  may produce at each output; (4) how many slots are available at each output B&R block of  $A$ . For (1), we will assume that these numbers depend on the current state of  $A$ : thus,  $A$  knows how many tokens it needs to consume at the beginning of each firing (but the numbers may change from one firing to the next). For (3), we will assume that we know what is the maximum number of tokens that a block can produce at each of its ports. Then, we can transform any dynamic data flow diagram that satisfies these assumptions to an SBD, as illustrated in Figure 6.



**Figure 6.** Transformation of a dynamic data flow diagram into a synchronous block diagram.

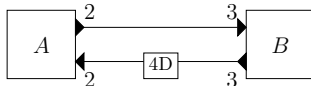
The figure shows a portion of the transformation, for a single block  $A$ . The interface of  $A$  needs to be modified into  $A'$  as previously. Also, the B&R blocks receive the control signals, as well as the triggers, as previously. Notice that the control signals cannot generally be eliminated in this case, since the sequence of operations applied to the B&R blocks is not statically known. Additionally, there is a feedback signal from the B&R blocks to the dynamic triggering block. This signal carries information of the types (2) and (4) listed above. Finally, there is a feedback signal

from each block to the triggering block, denoted “no. tokens” in the figure, which carries information of types (1) and (3) listed above.

## 6 Handling cyclic SDF diagrams

For simplicity, we discussed transformation of acyclic data flow diagrams in the above. Our method can be extended to cyclic SDF diagrams without major issues. It is worth noting, however, that when we transform cyclic SDF diagrams we may generate cyclic SBDs. This may happen only when the triggering policy is such that it fires multiple blocks simultaneously.

For example, consider the SDF diagram of Figure 7. The notation “4D” stands for “4 delays” and signifies that there are four initial tokens placed on the link from  $B$  to  $A$ . Suppose we are given the schedule  $A; A; B; A; B; A; A; B; A; B$  and we somehow generate the triggering policy  $A; (A, B); A; (B, A); A; B; A; B$ . Then, during the triggering step  $(A, B)$  there is feed-through from  $A$  to  $B$ , while during the step  $(B, A)$  there is feed-through from  $B$  to  $A$ . In a static block diagram such as an SBD, this manifests as a static dependency cycle between blocks. Notice, however, that there is no true dependency cycle, since the two dependencies never occur simultaneously (i.e., at the same triggering step). Thus, the SBD is “constructive” in the sense of [8, 1], that is, it has a well-defined semantics that can be computed during simulation.



**Figure 7.** A cyclic SDF diagram.

## 7 Conclusions

We have presented a method to automatically transform SDF diagrams into SBDs, such that the stream semantics of SDF is preserved. This shows that tools such as Simulink can be used to capture and simulate SDF models. We also discussed how the transformation can be extended to dynamic data flow diagrams, where the number of tokens produced and consumed is generally determined at run-time.

## Acknowledgments

We would like to thank Maarten Wiggers for triggering this work and for helpful discussions.

## References

- [1] G. Berry. The Constructive Semantics of Pure Esterel, 1999.
- [2] S. Bhattacharyya, E. Lee, and P. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.
- [3] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.
- [4] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, Nov. 1966.
- [5] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
- [6] R. Lubliner and S. Tripakis. Modular Code Generation from Triggered and Timed Block Diagrams. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’08)*. IEEE CS Press, Apr. 2008.
- [7] R. Lubliner and S. Tripakis. Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams. In *Design, Automation, and Test in Europe (DATE’08)*. ACM, Mar. 2008.
- [8] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, 1994.
- [9] R. Reiter. Scheduling parallel computations. *J. ACM*, 15(4):590–599, 1968.
- [10] M. H. Wiggers, M. J. Bekooij, and G. J. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *RTAS’08*, pages 183–194. IEEE Computer Society, 2008.