Chapter 2
Data Flow Modeling and Transformation

2.1 Introducing Data Flow Graphs

By nature, hardware is parallel and software is sequential. As a result, software models (C programs) are not very well suited to capture hardware implementations, and vice versa, hardware models (RTL programs) are not a good abstraction to describe software. However, designers frequently encounter situations for which a given design may use either hardware or software as a target. Trying to do both (writing a full C program *as well as* a full hardware design) is not an option; it requires the designer to work twice as hard. An alternative is to use a high-level model, which enables the designer to express a design without committing to a hardware or a software implementation. Using a high-level model, the designer can gain further insight into the specification, and decide on the right path for implementation.

In the design of signal processing systems, the need for modeling is well known. Signal processing engineers describe complex systems, such as digital radios and radar processing units, using *block diagrams*. A block diagram is a high-level representation of the target system as a collection of smaller functions. A block diagram does not specify if a component should be implemented as hardware or software; it only expresses the operations performed on data signals. We are specifically interested in *digital* signal processing systems. Such systems represent signals as streams of discrete samples rather than continuous waveforms.

Figure 2.1a shows the block diagram for a simple digital signal processing system. It's a pulse-amplitude modulation (PAM) system, and it is used to transmit digital information over bandwidth-limited channels. A PAM signal is created from binary data in two steps. First, each word in the file needs to be mapped to PAM symbols, an alphabet of pulses of different heights. An entire file of words will thus be converted to a stream of PAM symbols or pulses. Next, the stream of pulses needs to be converted to a smooth shape using pulse-shaping. Pulse-shaping ensures that the bandwidth of the resulting signal does not exceed the PAM symbol rate. For example, if a window of 1,000 Hz transmission bandwidth is available, then we

P.R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*,
DOI 10.1007/978-1-4614-3737-6_2,

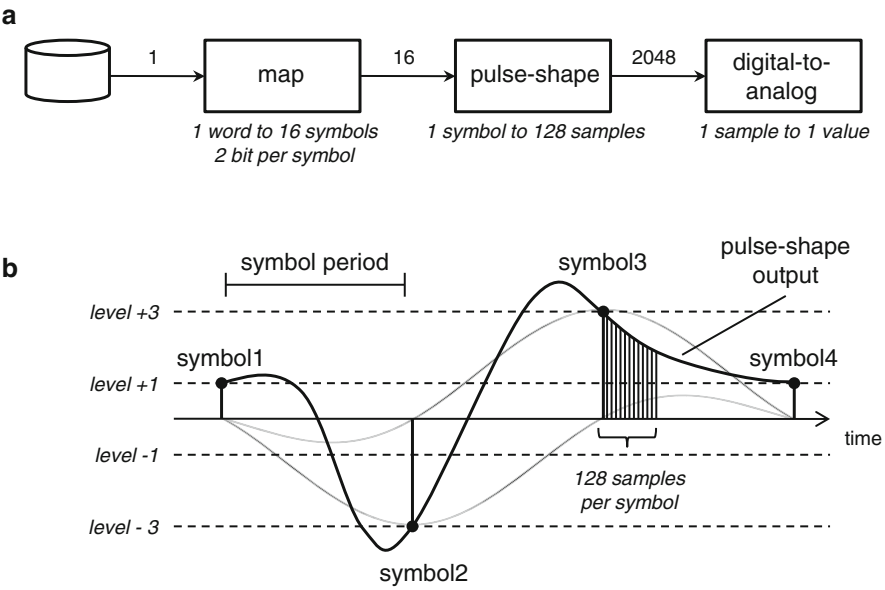


Fig. 2.1 (**a**) Pulse-amplitude modulation system. (**b**) Operation of the pulse-shaping unit

can transmit 1,000 PAM symbols per second. In a digital signal processing system, a *smooth* curve is achieved by *oversampling*: calculating many closely-spaced discrete samples. The output of the pulse-shaping unit produces many samples for each input symbol pulse, but it is still a stream of discrete samples. The final module in the block diagram is the digital-to-analog converter, which will convert the stream of discrete samples into a continuous signal.

Figure 2.1a shows a PAM-4 system, which uses four different symbols. Since there are four different symbols, each PAM symbol holds 2 bits of source information. A 32-bit word from a data source is encoded with 16 PAM-4 symbols. The first block in the PAM transmission system makes the conversion of a single word to a sequence of 16 PAM-4 symbols. Figure 2.1b shows that each PAM-4 symbol is mapped to a pulse with four possible signal levels: $\{-3, -1, 1, 3\}$. Once the PAM-4 signals are available, they are shaped to a smooth curve using a pulse-shape filter. The input of this filter is a stream of symbol pulses, while the output is a stream of samples at a much higher rate. In this case, we generate 128 samples for each symbol.

Figure 2.1b illustrates the operation of the pulse-shape filter. The smooth curve at the output of the pulse-shape filter connects the top of each pulse. This is achieved by an interpolation technique, which extends the influence of a single symbol pulse over many symbol periods. The figure illustrates two such interpolation curves, one for `symbol2` and one for `symbol3`. The pulse-shape filter will produce 128 samples for each symbol entered into the pulse-shape filter.

Listing 2.1 C example

```
1  extern int read_from_file();
2  extern int map_to_symbol(int, int);
3  extern int pulse_shape(int, int);
4  extern void send_to_da(int);
5
6  int main() {
7    int word, symbol, sample;
8    int i, j;
9    while (1) {
10     word = read_from_file();
11     for (i=0; i<16; i++) {
12        symbol = map_to_symbol(word, i);
13        for (j=0; j<128; j++)
14          sample = pulse_shape(symbol, j);
15          send_to_da(sample);
16     }
17   }
18 }
```

Now let's consider the construction of a simulation model for this system. We focus on capturing its functionality, and start with a C program as shown in Listing 2.1. We will ignore the implementation details of the function calls for the time being, and only focus on the overall structure of the program.

The program in Listing 2.1 is fine as a system simulation. However, as a model for the implementation, this C program is too strict, since it enforces *sequential* execution of all functions. If we observe Fig. 2.1a carefully, we can see that the block diagram does not require a sequential execution of the symbol mapping function and the pulse shaping function. The block diagram *only* specifies the flow of data in the system, but not the execution order of the functions. The distinction is subtle but important. For example, in Fig. 2.1a, it is possible that the `map` module and the `pulse-shape` module work in parallel, each on a different symbol. In Listing 2.1 on the other hand, the `map_to_symbol()` function and the `pulse_shape()` function will *always* execute sequentially. In hardware-software codesign, the implementation target could be either parallel or else sequential. The program in Listing 2.1 favors a sequential implementation, but it does not encourage a parallel implementation in the same manner as a block diagram.

This illustrates how the selection of a modeling technique can constrain the solutions that may be achieved starting from that model. In general, building a sequential implementation for a parallel model (such as a block diagram) is much easier than the opposite – building a parallel implementation from a sequential model. Therefore, we favor modeling styles that enable a designer to express parallel activities at the highest level of abstraction.

In this chapter we will discuss a modeling technique, called Data Flow, which achieves the objective of a parallel model. Data Flow models closely resemble

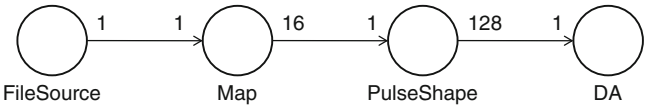


Fig. 2.2 Data flow model for the pulse-amplitude modulation System

block diagrams. The PAM-4 system, as a Data Flow model, is shown in Fig. 2.2. In this case, the different functions of the system are mapped as individual entities or *actors* such as `FileSource`, `Map`, `PulseShape` and `DA`. These actors are linked through communication channels or *queues*. The inputs and outputs of each actor are marked with the relative rate of communications. For example, there are 16 samples produced by `Map` for each input sample. Each actor is an *independent unit*, continuously checking its input for the availability of data. As soon as data appears, each actor will calculate the corresponding output, passing the result to the next actor in the chain. In the remainder of this chapter, we will discuss the precise construction details of data flow diagrams. For now, we only point out the major differences of this modeling style compared to modeling in C.

- A strong point of Data Flow models, and the main reason why signal processing engineers love to use them, is that a Data Flow model is a *concurrent* model. Indeed, the actors in Fig. 2.2 operate and execute as individual concurrent entities. A concurrent model can be mapped to a parallel or a sequential implementation, and so they can target hardware as well as software.
- Data Flow models are distributed, and there is no need for a central controller or 'conductor' in the system to keep the individual system components in pace. In Fig. 2.2, there is no central controller that tells the actors when to operate; each actor can determine for itself when it's time to work.
- Data Flow models are modular. We can develop a design library of data flow components and then use that library in a plug-and-play fashion to construct data flow systems.
- Data Flow models can be analyzed. Certain properties, such as their ability to avoid deadlock, can be determined directly from the design. Deadlock is a condition sometimes experienced by real-time computing systems, in which the system becomes unresponsive. The ability to analyze the behavior of models at high abstraction level is an important advantage. C programs, for example, do not offer such convenience. In fact, a C designer typically determines the correctness of a C program by running it, rather than analyzing it!

Data Flow has been around for a surprisingly long time, yet it has been largely overshadowed by the stored-program (Von Neumann) computing model. Data Flow concepts have been explored since the early 1960s. By 1974, Jack Dennis had developed a language for modeling data flow, and described data flow using graphs, similar to our discussion in this chapter. In the 1970s and 1980s, an active research community was building not only data flow-inspired programming languages and

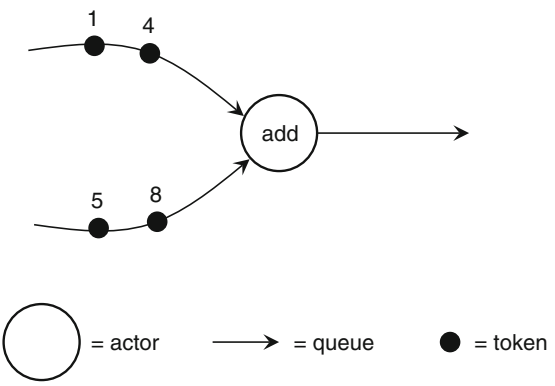


Fig. 2.3 Data flow model of an addition

tools, but also computer architectures that implement data flow computing models. The work from Arvind was seminal in this area, resulting in several different computer architectures and tools (see Further Reading at the end of this chapter).

Today, data flow remains very popular to describe signal processing systems. For example, commercial tools such as Simulink® are based on the ideas of data flow. A interesting example of an academic environment is the Ptolemy project at UC Berkeley (http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm). The Ptolemy design environment can be used for many different types of system specification, including data flow. The examples on the website can be run inside of a web browser as Java applets.

In the following sections, we will consider the elements that make up a data flow model. We will next discuss a particular class of data flow models called Synchronous Data Flow Graphs (SDF). We will show how SDF graphs can be formally analyzed. Later, we will discuss transformations on SDF graphs, and show how transformations can lead to better, faster implementations.

2.1.1 Tokens, Actors, and Queues

Figure 2.3 shows the data flow model of a simple addition. This model contains the following elements.

- *Actors* contain the actual operations. Actors have a bounded behavior (meaning that they have a precise beginning and ending), and they iterate that behavior from start to completion. One such iteration is called an actor firing. In the example above, each actor firing would perform a single addition.
- *Tokens* carry information from one actor to the other. A token has a value, such as '1', '4', '5' and '8' in Fig. 2.3.

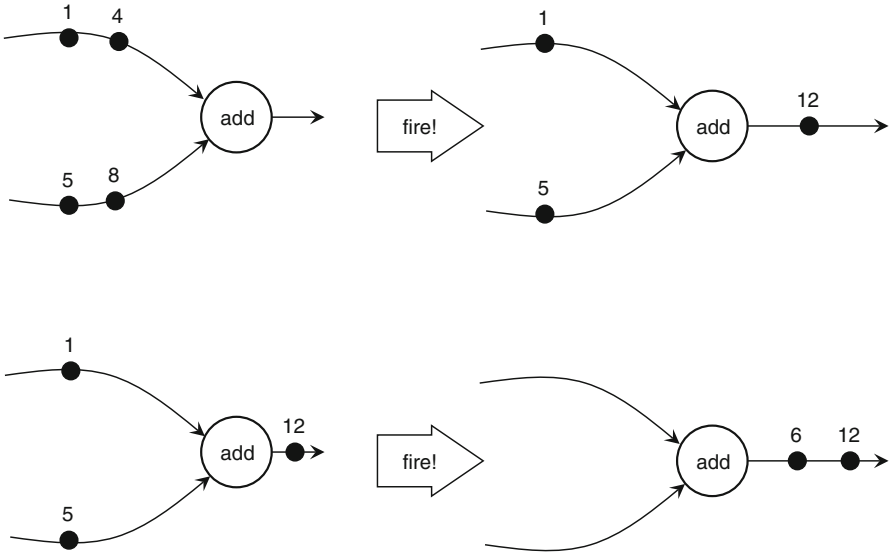


Fig. 2.4 The result of two firings of the add actor, each resulting in a different marking

- *Queues* are unidirectional communication links that transport tokens from one actor to the other. Data Flow queues have an infinite amount of storage, so that tokens will never get lost in a queue. Data Flow queues are first-in first-out. In Fig. 2.4, there are two tokens in the upper queue, one with value '1' and one with value '4'. The '4' token was entered first into the queue, the '1' token was entered after that. When the 'add' actor will read a token from that queue, the actor will first read the token with value '4' and next the token with value '1'.

When a data flow model executes, the actors will read tokens from their input queues, read the value of these tokens and calculate the corresponding output value, and generate new tokens on their output queues. Each single execution of an actor is called the firing of that actor. Data flow execution then is expressed as a sequence of (possibly concurrent) actor firings.

Conceptually, data flow models are untimed. The firing of an actor happens instantaneously, although any real implementation of an actor does require a finite amount of time. *Untimed* does not mean zero time; it only means that time is irrelevant for data flow models. Indeed, in data flow, the execution is guided by the presence of data, not by a program counter or by a clock signal. An actor will never fire if there's no input data, but instead it will wait until sufficient data is available at its inputs.

A graph with tokens distributed over queues is called a *marking* of a data flow model. When a data flow model executes, the entire graph goes through a series of markings that drive data from the inputs of the data flow model to

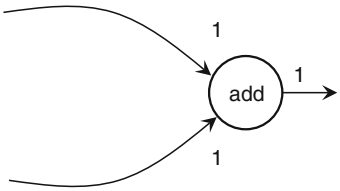


Fig. 2.5 Data flow actor with production/consumption rates

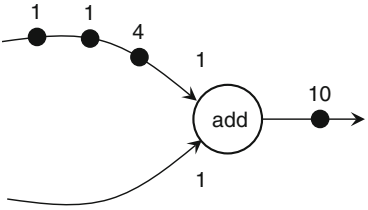


Fig. 2.6 Can this model do any useful computations?

Fig. 2.7 Example of a multi-rate data flow model

the outputs. Each marking corresponds to a different state of the system, and the execution of a data flow model is defined by a sequence of markings. To an external observer, the marking (i.e., the distribution of the tokens on the queues) is the only observable state in the system. This is a crucial observation! It implies that an actor cannot use internal state variables that would affect the execution of the system, and thus the marking sequence. If we want to model system state that would affect the execution, we need to express it using tokens.

2.1.2 Firing Rates, Firing Rules, and Schedules

When should an actor fire? The *firing rule* describes the necessary and sufficient conditions for an actor to fire. Simple actors such as the add actor can fire when there is a single token on each of its queues. A firing rule thus involves testing the number of tokens on each of its input queues. The required number of tokens can be annotated to the actor input. Similarly, the amount of tokens that an actor produces per firing can be annotated to the actor output. These numbers are called the token consumption rate (at the actor inputs) and token production rate (at the actor outputs). The production/consumption rates of the add actor could be written such as shown in Figs. 2.5 or 2.6.

Data Flow actors may consume or produce more than one token per actor firing. Such models are called multirate data flow models. For example, the actor in Fig. 2.7

has a consumption rate of 2 and a production rate of 1. It will consume two tokens per firing from its input, add them together, and produce an output token as result.

2.1.3 Synchronous Data Flow (SDF) Graphs

When the number of tokens consumed/produced per actor firing is a fixed and constant value, the resulting class of systems is called synchronous data flow or SDF. The term synchronous means that the token production and consumption rates are known, fixed numbers. SDF semantics are not universal. For example, not every C program can be translated to an equivalent SDF graph. Data-dependent execution cannot be expressed as an SDF graph: data-dependent execution implies that actor firing is defined not only by the presence of tokens, but also by their value.

Nevertheless, SDF graphs have a significant advantage: their properties can be analyzed mathematically. The structure of an SDF graph, and the production/consumption rates of tokens on the actors, determines if a feasible schedule of actor firings is possible. We will demonstrate a technique that can analyze an SDF graph, and derive such a feasible schedule, if it exists.

2.1.4 SDF Graphs are Determinate

Assuming that each actor implements a deterministic function, then the entire SDF execution is determinate. This means that results, computed using an SDF graph, will always be the same. This property holds regardless of the firing order (or schedule) of the actors. Figure 2.8 illustrates this property. This graph contains actors with unit production/consumption rates. One actor adds tokens, the second actor increments the value of tokens. As we start firing actors, tokens are transported through the graph. After the first firing, an interesting situation occurs: both the add actor as well as the plus1 actor can fire. A first case, shown on the left of Fig. 2.8, assumes that the plus1 actor fires first. A second case, shown on the right of Fig. 2.8, assumes that the add actor fires first. However, regardless what path is taken, the graph marking eventually converges to the result shown at the bottom.

Why is this property so important? Assume that the add actor and the plus1 actor execute on two different processors, a slow one and a fast one. Depending upon which actor runs on the fast processor, the SDF execution will follow the left marking or else the right marking of the figure. Since SDF graphs are determinate, it doesn't matter which processor executes what actor: the results will be always the same. In other words, an SDF system will work as specified, regardless of the technology used to implement it. Of course, actors must be completely and correctly implemented, firing rule and all. Determinate behavior is vital in many embedded applications, especially in applications that involve risk.

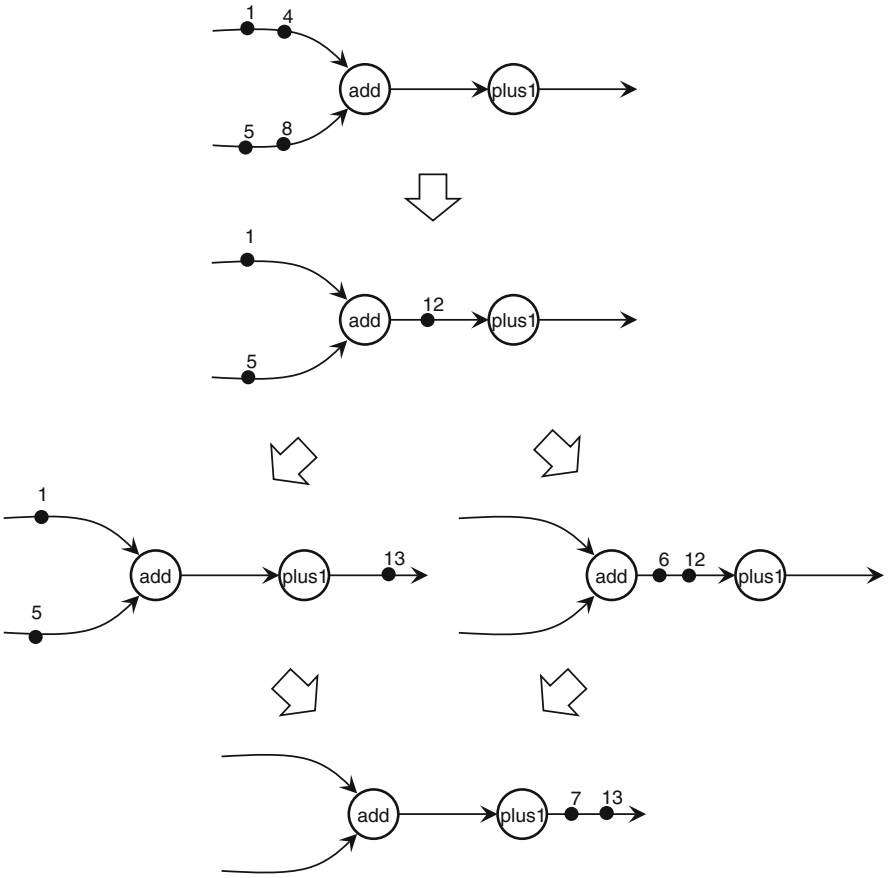


Fig. 2.8 SDF graphs are determinate

2.2 Analyzing Synchronous Data Flow Graphs

An admissible schedule for an SDF graph is one that can run forever without causing deadlock and without overflowing any of the communication queues. The term *unbounded execution* is used to indicate that a model runs forever; the term *bounded buffer* is used to indicate that no communication queue needs infinite depth. A deadlock situation occurs when the SDF graph ends up in a marking in which it is no longer possible to fire any actor.

Figure 2.9 shows two SDF graphs where these two problems are apparent. Which graph will deadlock, and which graph will result in an infinite amount of tokens?

Given an arbitrary SDF graph, it is possible to test if it is free from deadlock, and if it only needs bounded storage under unbounded execution. The nice thing about this test is that we don't need to use any simulation; the test can be done using basic

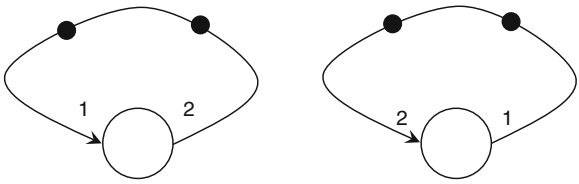


Fig. 2.9 Which SDF graph will deadlock, and which is unstable?

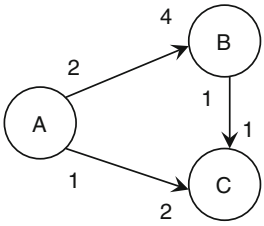


Fig. 2.10 Example SDF graph for PASS construction

matrix operations. We will study the method of Lee to create so-called Periodic Admissible Schedules (PASS). A PASS is defined as follows.

- A schedule is the order in which the actors must fire.
- An admissible schedule is a firing order that will not cause deadlock and that yields bounded storage.
- Finally, a periodic admissible schedule is a schedule that is suitable for unbounded execution, because it is periodic (meaning that after some time, the same marking sequence will repeat). We will consider Periodic Admissible Sequential Schedules, or PASSs for short. Such a sequential schedule requires only a single actor to fire at a time. A PASS would be used, for example, to execute an SDF model on a microprocessor.

2.2.1 Deriving Periodic Admissible Sequential Schedules

We can create a PASS for an SDF graph (and test if one exists) with the following four steps.

1. Create the topology matrix G of the SDF graph;
2. Verify the rank of the matrix to be one less than the number of nodes in the graph;
3. Determine a firing vector;
4. Try firing each actor in a round robin fashion, until it reaches the firing count as specified in the firing vector.

We will demonstrate each of these steps using the example of the three-node SDF graph shown in Fig. 2.10.

Step 1. Create a topology matrix for this graph. This topology matrix has as many rows as graph edges (FIFO queues) and as many columns as graph nodes. The entry (i, j) of this matrix will be positive if the node j produces tokens into graph edge i. The entry (i, j) will be negative if the node j consumes tokens from graph edge i. For the above graph, we thus can create the following topology matrix. Note that G does not have to be square – it depends on the amount of queues and actors in the system.

$$G = \begin{bmatrix} 2 & -4 & 0 \\ 1 & 0 & -2 \\ 0 & 1 & -1 \end{bmatrix} \begin{matrix} \leftarrow edge(A,B) \\ \leftarrow edge(A,C) \\ \leftarrow edge(B,C) \end{matrix} \tag{2.1}$$

Step 2. The condition for a PASS to exist is that the rank of G has to be one less than the number of nodes in the graph. The proof of this theorem is beyond the scope of this book, but can be consulted in (Lee and Messerschmitt 1987). The rank of a matrix is the number of independent equations in G. It can be verified that there are only two independent equations in G. For example, multiply the first column with -2 and the second column with -1, and add those two together to find the third column. Since there are three nodes in the graph and the rank of G is 2, a PASS is possible.

Step 2 verifies that tokens cannot accumulate on any of the edges of the graph. We can find the resulting number of tokens by choosing a firing vector and making a matrix multiplication. For example, assume that A fires two times, and B and C each fire zero times. This yields the following firing vector:

$$q = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \tag{2.2}$$

The residual tokens left on the edges after these firings are two tokens on $edge(A,B)$ and a token on $edge(A,C)$:

$$b = Gq = \begin{bmatrix} 2 & -4 & 0 \\ 1 & 0 & -2 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \tag{2.3}$$

Step 3. Determine a periodic firing vector. The firing vector indicated above is not a good choice to obtain a PASS: each time this firing vector executes, it adds three tokens to the system. Instead, we are interested in firing vectors that leave no additional tokens on the queues. In other words, the result must equal the zero-vector.

$$Gq_{PASS} = 0 \tag{2.4}$$

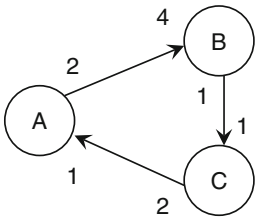


Fig. 2.11 A deadlocked graph

Since the rank of G is less than the number of nodes, this system has an infinite number of solutions. Intuitively, this is what we should expect. Assume a firing vector (a,b,c) would be a solution that can yield a PASS, then also $(2a,2b,2c)$ will be a solution, and so is $(3a,3b,3c)$, and so on. You just need to find the simplest one. One possible solution that yields a PASS is to fire A twice, and B and C each once:

$$q_{PASS} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} \tag{2.5}$$

The existence of a PASS firing vector does not guarantee that a PASS will also exist. For example, just by changing the direction of the (A,C) edge, you would still find the same q_{PASS}, but the resulting graph is deadlocked since all nodes are waiting for each other. Therefore, there is still a fourth step: construction of a valid PASS.

Step 4. Construct a PASS. We now try to fire each node up to the number of times specified in q_{PASS}. Each node which has the adequate number of tokens on its input queues will fire when tried. If we find that we can fire no more nodes, and the firing count of each node is less than the number specified in q_{PASS}, the resulting graph is deadlocked.

We apply this on the original graph and using the firing vector $(A = 2, B = 1, C = 1)$. First we try to fire A, which leaves two tokens on (A,B) and one on (A,C). Next, we try to fire B – which has insufficient tokens to fire. We also try to fire C but again have insufficient tokens. This completes our first round through – A has fired already one time. In the second round, we can fire A again (since it has fired less than two times), followed by B and C. At the end of the second round, all nodes have reached the firing count specified in the PASS firing vector, and the algorithm completes. The PASS we are looking for is (A,A,B,C).

The same algorithm, when applied to the deadlocked graph in Fig. 2.11, will immediately abort after the first iteration, because no node was able to fire.

Note that the determinate property of SDF graphs implies that we can try to fire actors in any order of our choosing. So, instead of trying the order (A,B,C) we can also try (B,C,A). In some SDF graphs (but not in the one discussed above), this may lead to additional PASS solutions.

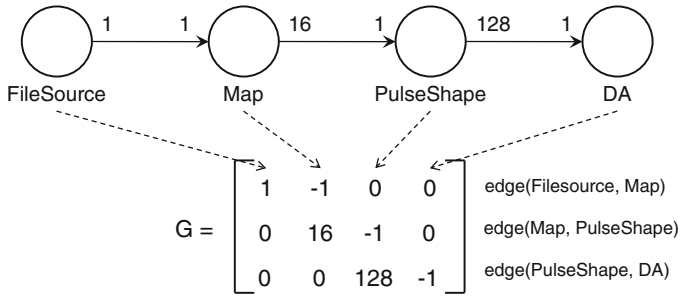


Fig. 2.12 Topology matrix for the PAM-4 system

2.2.2 Example: Deriving a PASS for the PAM-4 System

At the start of this chapter, we discussed the design of a digital pulse-amplitude modulation system for the generation of PAM4 signals. The system, shown in Fig. 2.12, is modeled with four data flow actors: a word source, a symbol mapper, a pulse shaper, and a digital-to-analog converter. The system is a multi-rate data flow system. For every byte modulated, $16.128 = 2048$ output samples are generated.

Our objective is to derive a PASS for this data flow system. The first step is to derive the topology matrix G. The matrix has three rows and four columns, corresponding to the three queues and four actors in the system. The second step in deriving the PASS is to verify that the rank of this topology matrix equals the number of data flow actors minus one. It's easy to demonstrate that G indeed consists of three independent equations: no row can be created as a linear combination of two others. Hence, we confirm that the condition for a PASS to exist is fulfilled. Third, we have to derive a feasible firing vector for this system. This firing vector, q_{PASS}, needs to yield a zero-vector when multiplied with the topology matrix. The solution for q_{PASS} is to fire the `Filesource` and `Map` actors one time, the `PulseShape` actor 16 times, and the `DA` actor 2,048 times.

$$G.q_{PASS} = \begin{bmatrix} +1 & -1 & 0 & 0 \\ 0 & 16 & -1 & 0 \\ 0 & 0 & 128 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 16 \\ 2{,}048 \end{bmatrix} = 0 \tag{2.6}$$

The final step is to derive a concrete schedule with the derived firing rates. We discuss two alternative solutions.

- By inspection of the graph in Fig. 2.12, we conclude that firing the actors from left to right according to their q_{PASS} firing rate will result in a feasible solution that ensures sufficient tokens in each queue. Thus, we start by firing `FileSource` once, followed by `Map`, followed by 16 firings of `PulseShape`, and finally 2,048 firings of `DA`. This particular schedule will require a FIFO of 16 positions

for the middle queue in the system, and a FIFO of 2,048 positions for the rightmost queue in the system.
- An alternative schedule is to start firing `FileSource` once, followed by `Map`. Next, the following sequence is repeated 16 times: fire `PulseShape` once, followed by 128 firings of `DA`. The end result of this alternate schedule is identical to the first schedule. However, the amount of intermediate storage is much lower: the rightmost queue in the system will use at most 128 positions.

Hence, we conclude that the concrete schedule in a PASS affects the amount of storage used by the communication queues. Deriving an optimal interleaving of the firings is a complex problem in itself (See Further Reading).

This completes our discussion of PASS. SDF has very powerful properties, which enable a designer to predict critical system behavior such as determinism, deadlock, and storage requirements. Yet, SDF is not a universal specification mechanism; it is not a good replacement for any type of application. The next part will further elaborate on the difficulty of implementing control-oriented systems using data flow modeling.

2.3 Control Flow Modeling and the Limitations of Data Flow Models

SDF systems are distributed, data-driven systems. They execute whenever there is data to process, and remain idle when there is nothing to do. However, SDF seems to have trouble to model control-related aspects. Control appears in many different forms in system design, for example:

- **Stopping and restarting**. An SDF model never terminates; it just keeps running. Stopping and re-starting is a control-flow property that cannot be addressed well with SDF graphs.
- **Mode-switching**. When a cell-phone switches from one standard to the other, the processing (which may be modeled as an SDF graph) needs to be reconfigured. However, the topology of an SDF graph is fixed and cannot be modified at runtime.
- **Exceptions**. When catastrophic events happen, processing may suddenly need to be altered. SDF cannot model exceptions that affect the entire graph topology. For example, once a token enters a queue, the only way of removing it is to read the token out of the queue. It is not possible to suddenly flush the queue on a global, exceptional condition.
- **Run-time Conditions**. A simple if-then-else statement (choice between two activities depending on an external condition) is troublesome for SDF. An SDF node cannot simply 'disappear' or become inactive – it is always there. Moreover, we cannot generate conditional tokens, as this would violate SDF rules which require fixed production/consumption rates. Thus, SDF cannot model conditional execution such as required for if-then-else statements.

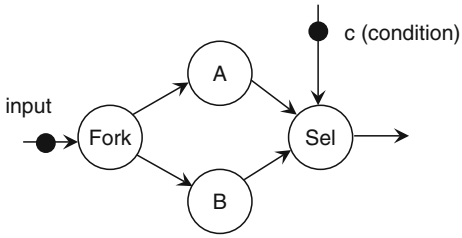


Fig. 2.13 Emulating if-then-else conditions in SDF

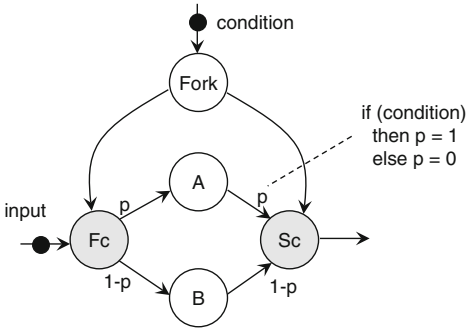


Fig. 2.14 Implementing if-then-else using Boolean Data Flow

There are two solutions to the problem of control flow modeling in SDF. The first one is to emulate control flow using SDF, at the cost of some modeling overhead. The second one is to extend the semantics of SDF. We give a short example of each strategy.

2.3.1 Emulating Control Flow with SDF Semantics

Figure 2.13 shows an example of an if-then-else statement, SDF-style. Each of the actors in the above graph are SDF actors. The last one is a selector-actor, which will transmit either the A or B input to the output depending on the value of the input condition. Note that when Sel fires, it will consume a token from each input, so both A and B have to run for each input token. This is thus not really an if-then-else in the same sense as in C programming. The approach taken by this graph is to implement both the if-leg and the else-leg and afterwards transmit only the required result. This approach may work when there is sufficient parallelism available. For example, in hardware design, the equivalent of the Sel node would be a multiplexer.

2.3.2 Extending SDF Semantics

Researchers have also proposed extensions on SDF models. One of these extensions was proposed by Joseph Buck, and is called BDF (Boolean Data Flow) (Lee and Seshia 2011). The idea of BDF is to make the production and consumption-rate of a

token dependent on the value of an external control token. In Fig. 2.14, the condition token is distributed over a fork to a conditional fork and a conditional merge node. These conditional nodes are BDF.

- The conditional fork will fire when there is an input token and a condition token. Depending on the value of the condition token, it will produce an output on the upper or the lower queue. We used a conditional production rate p to indicate this. It is impossible to determine the value of p upfront – this can only be done at runtime.
- The conditional merge will fire when there is a condition token. If there is, it will accept a token from the upper input or the lower input, depending on the value of the condition. Again, we need to introduce a conditional consumption rate.

The overall effect is that either node A or else node B will fire, but never both. Even a simple extension on SDF already takes jeopardizes the basic properties which we have enumerated above. For example, a consequence of using BDF instead of SDF is that we now have data flow graphs that are only conditionally admissible. Moreover, the topology matrix now will include symbolic values (p), and becomes harder to analyze. For five conditions, we would have to either analyze a matrix with five symbols, or else enumerate all possible condition values and analyze 32 different matrices (each of which can have a different series of markings). In other words, while BDF can help solving some of practical cases of control, it quickly becomes impractical for analysis.

Besides BDF, researchers have also proposed other flavors of control-oriented data flow models, such as Dynamic Data Flow (DDF) which allows variable production and consumption rates, and Cyclo-Static Data Flow (CSDF) which allows a fixed, iterative variation on production and consumption rates. All of these extensions break down the elegance of SDF graphs to some extent. SDF remains a very popular technique for Digital Signal Processing applications. But the use of BDF, DDF and the like has been limited.

2.4 Adding Time and Resources

So far, we have treated data flow graphs as untimed: the analysis of data flow graphs was based only on their marking (distribution of tokens), and not on the time needed to complete a computation. However, we can also use the data flow model to do performance analysis. By introducing a minimal resource model (actor execution time and bounded FIFO queues), we can analyze the system performance of a data flow graph. Furthermore, we can analyze the effect of performance-enhancing transformations on the data flow graph.

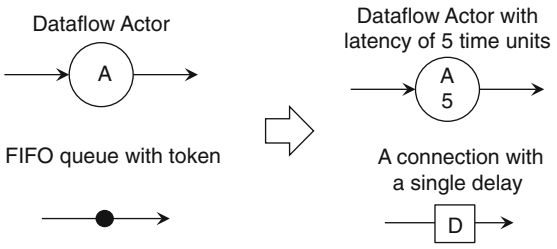


Fig. 2.15 Enhancing the SDF model with resources: execution time for actors, and delays for FIFO queues

2.4.1 Real-Time Constraints and Input/Output Sample Rate

A data flow graph is a model for a repeating activity. For example, the PAM-4 modulation system described in this chapter's introduction transforms an infinite stream of input samples (words) into an infinite stream of output samples. The model shows how one single sample is processed, and the streaming character is implicit.

The input sample rate is the time period between two adjacent input-samples from the stream. The sample rate typically depends on the application. CD audio samples, for example, are generated at 44,100 samples per second. The input sample rate thus sets a design constraint for the performance of the data flow system: it specifies how quickly the data flow graph must be computed in order to achieve real-time performance. A similar argument can be made for the output sample rate. In either case, when there's a sample-rate involved with the input or the output of a data flow graph, there is also a real-time constraint on the computation speed for the data flow graph.

We define the input *throughput* as the amount of input samples per second. Similarly, we define the output throughput as the amount of output samples per second. The *latency* is the time required to process a single token from input to output. Throughput and latency are two important system constraints.

2.4.2 Data Flow Resource Model

In this section, we are interested in performance analysis of data flow graphs. This requires the introduction of time and resources. Figure 2.15 summarizes the two enhancements needed.

- Every actor is decorated with an execution latency. This is the time needed by the actor to complete a computation. We assume that the actor's internal program requires all inputs to be available at the start of the execution. Similarly, we assume that the actor's internal program produces all outputs simultaneously after the actor latency. Latency is expressed in time units, and depending on the implementation target, a suitable unit can be chosen – clock cycles, nanoseconds, and so on.

- Every FIFO queue is replaced with a communication channel with a fixed number of delays. A delay is a storage location that can hold one token. A single delay can hold a token for a single actor execution. Replacing a FIFO queue with delay storage locations also means that the actor firing rule needs to be changed. Instead of testing the number of elements in a FIFO queue, the actor will now test for the presence of a token in a delay storage location.

The use of a data flow resource model enables us to analyze how fast a data flow graph will run. Figure 2.16 shows three single-rate data flow graphs, made with two actors *A* and *B*. Actor *A* needs five units of latency, while actor *B* requires three units of latency. This data flow graph also has an input and an output connection, through which the system can accept a stream of input samples, and deliver a stream of output samples. For our analysis, we do not define an input or output sample rate. Instead, we are interested to find out how fast these data flow graphs can run.

The easiest way to analyze this graph is to evaluate the latency of samples as they are processed through the data flow graph. Eventually, this analysis yields the time instants when the graph reads from the system input, or writes to the system output.

In the graphs of Fig. 2.16a, b, there is a single delay element in the loop. Data input/output is defined by the combined execution time of actor *A* and actor *B*. The time stamps for data production for the upper graph and the middle graph are different because of the position of the delay element: for the middle graph, actor *B* can start at system initialization time, since a token is available from the delay element.

In the graph of Fig. 2.16c, there are two delay elements in the loop. This enables actor *A* and actor *B* to operate in parallel. The performance of the overall system is defined by the slowest actor *A*. Even though actor *B* completes in three time units, it needs to wait for the next available input until actor *A* has updated the delay element at its output.

Hence, we conclude that the upper two graphs have a throughput of 1 sample per 8 time units, and that the lower graph has a throughput of 1 sample per 5 time units.

2.4.3 Limits on Throughput

The example in the previous section illustrates that the number of delays, and their distribution over the graph, affects the throughput of the data flow system. A second factor that affects the latency are the feedback links or loops in the data flow graph. Together, loops and delays determine an upper bound on the computation speed of a data flow graph.

We define two quantities to help us analyze the throughput limits of a data flow system: the *loop bound* and the *iteration bound*. The loop bound is the round-trip delay in a given loop of a data flow graph, divided by the number of delays in that loop. The iteration bound is the highest loop bound for any loop in a given data flow graph. The iteration bound sets an upper limit for the computational throughput of a data flow graph.

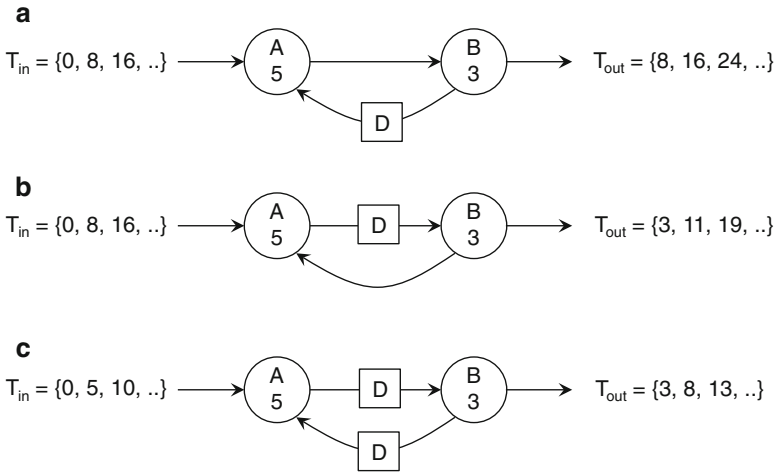


Fig. 2.16 Three data flow graphs

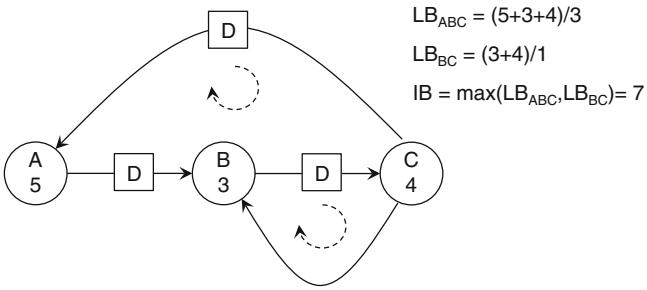


Fig. 2.17 Calculating loop bound and iteration bound

These concepts are explained with an example as shown in Fig. 2.17. This graph has three actors *A*, *B*, and *C*, each with different execution times. The graph has two loops: *BC* and *ABC*. The loop bounds are:

$$LB_{BC} = \frac{3+4}{1} = 7$$

$$LB_{ABC} = \frac{5+3+4}{3} = 4$$

The iteration bound of the system in Fig. 2.17 is the maximum of these two, or seven. This iteration bound implies that the implementation of this data flow graph will need at least 7 time units to process every iteration. The iteration bound thus sets an upper limit on throughput. If we inspect this graph closely, we conclude that loop *BC* is indeed the bottleneck of the system. Actors *A* and *C* have delays at their inputs, so that they can always execute in parallel. Actor *B* however, needs to wait

Fig. 2.18 Iteration bound for a linear graph

for a result from *C* before it can compute its output. Therefore, actor *B* and *C* can never run in parallel, and together, they define the iteration bound of the system.

It may appear that a linear graph, which has no loop, won't have an iteration bound. However, even a linear graph has an implicit iteration bound: a new sample cannot be read at the input until the output sample, corresponding to the previous input sample, has been generated. Figure 2.18 illustrates this for a linear graph with two actors *A* and *B*. When *A* reads a new sample, *B* will compute a corresponding output at time stamp 8. *A* can only start a new computation at time stamp 8; the absence of a delay between *A* and *B* prevents this any earlier. The analysis of the linear sections at the inputs and/or outputs of a data flow graph can be easily brought into account by assuming an implicit feedback from each output of a data flow graph to each input. This outermost loop is analyzed like any other loop to find the iteration bound for the overall system.

The iteration bound is an upper-limit for throughput. A given data flow graph may or may not be able to execute at its iteration bound. For example, the graph in Fig. 2.16c has an iteration bound of $(5+3)/2 = 4$ time units (i.e., a throughput of 1 sample per 4 time units), yet our analysis showed the graph to be limited at a throughput of 1 sample per 5 time units. This is because the slowest actor in the critical loop of Fig. 2.16c needs 5 time units to complete.

2.5 Transformations

Using performance analysis on data flow graphs, we can now evaluate suitable transformations to improve the performance of slow data flow graphs. We are interested in transformations that maintain the functionality of a data flow graph, but that increase the throughput and/or decrease the latency. This section will present several transformations which are frequently used to enhance system performance. Transformations don't affect the steady-state behavior of a data flow graph but, as we will illustrate, they may introduce transient effects, typically at startup. We cover the following transformations.

- **Multi-rate Expansion** is used to convert a multi-rate synchronous data flow graph to a single-rate synchronous data flow graph. This transformation is helpful because the other transformations assume single-rate SDF systems.
- **Retiming** considers the redistribution of delay elements in a data flow graph, in order to optimize the throughput of the graph. Retiming does not change the latency or the transient behavior of a data flow graph.

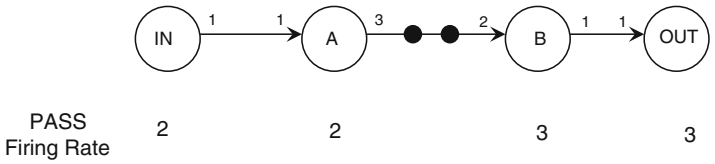


Fig. 2.19 Multi-rate data flow-graph

- **Pipeling** introduces additional delay elements in a data flow graph, with the intent of optimizing the iteration bound of the graph. Pipelining changes the throughput, and the transient behavior of a data flow graph.
- **Unfolding** increases the computational parallelism in a data flow graph by duplicating actors. Unfolding does not change the transient behavior of a data flow graph, but may modify the throughput.

2.5.1 Multirate Expansion

It is possible to transform a multi-rate SDF graph systematically to a single-rate SDF graph. The following steps to convert a multi-rate graph to a single-rate graph.

1. Determine the PASS firing rates of each actor
2. Duplicate each actor the number of times indicated by its firing rate. For example, given an actor A with a firing rate of 2, we create $A0$ and $A1$. These actors are two identical copies of the same generic actor A.
3. Convert each multi-rate actor input/output to multiple single-rate input/outputs. For example, if an actor input has a consumption rate of 3, we replace it with three single-rate inputs.
4. Re-introduce the queues in the data flow system to connect all actors. Since we are building a PASS system, the total number of actor inputs will be equal to the total number of actor outputs.
5. Re-introduce the initial tokens in the system, distributing them sequentially over the single-rate queues.

Consider the example of a multirate SDF graph in Fig. 2.19. Actor A produces three tokens per firing, actor B consumes two tokens per firing. The resulting firing rates are 2 and 3, respectively.

After completing steps 1–5 discussed above, we obtain the SDF graph shown in Fig. 2.20. The actors have duplicated according to their firing rates, and all multi-rate ports were converted to single-rate ports. The initial tokens are redistributed over the queues connecting instances of A and B. The distribution of tokens follows the sequence of queues between A,B (ie. follows the order a, b, etc.).

Multi-rate expansion is a convenient technique to generate a specification in which every actor needs to run at the same speed. For example, in a hardware

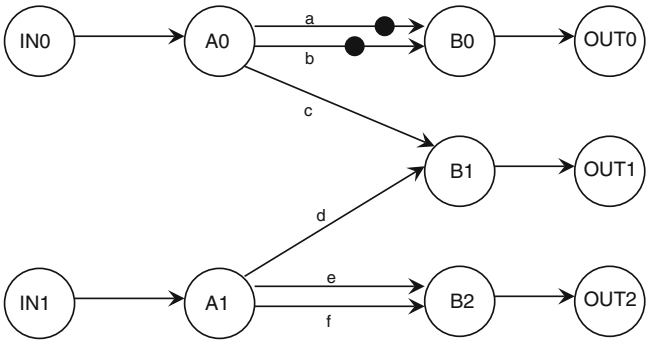


Fig. 2.20 Multi-rate SDF graph expanded to single-rate

implementation of data flow graphs, multi-rate expansion will enable all actors to run from the same clock signal.

2.5.2 Retiming

Retiming is a transformation on data flow graphs which doesn't change the total number of delays between input and output of a data flow graph. Instead, retiming is the redistribution the delays in the data flow graph. This way, the immediate dependency between actors can be broken, allowing them to operate in parallel. A retimed graph may have an increased system throughput. The retiming transformation is easy to understand. The transformation is obtained by evaluating the performance of successive markings of the data flow graph, and then selecting the one with the best performance.

Figure 2.21 illustrates retiming using an example. The top data flow graph, Fig. 2.21a, illustrates the initial system. This graph has an iteration bound of 8. However, the actual data output period of Fig. 2.21a is 16 time units, because actors *A*, *B*, and *C* need to execute as a sequence. If we imagine actor *A* to fire once, then it will consume the tokens (delays) at its inputs, and produce an output token. The resulting graph is shown in Fig. 2.21b. This time, the data output period has reduced to 11 time units. The reason is that actor *A* and the chain of actors *B* and *C*, can each operate in parallel. The graph of Fig. 2.21b is functionally identical to the graph of Fig. 2.21a: it will produce the same identical stream of output samples when given the same stream of input samples. Finally, Fig. 2.21c shows the result of moving the delay across actor *B*, to obtain yet another equivalent marking. This implementation is faster than the previous one; as a matter of fact, this implementation achieves the iteration bound of 8 time units per sample. No faster implementation exists for the given graph and the given set of actors.

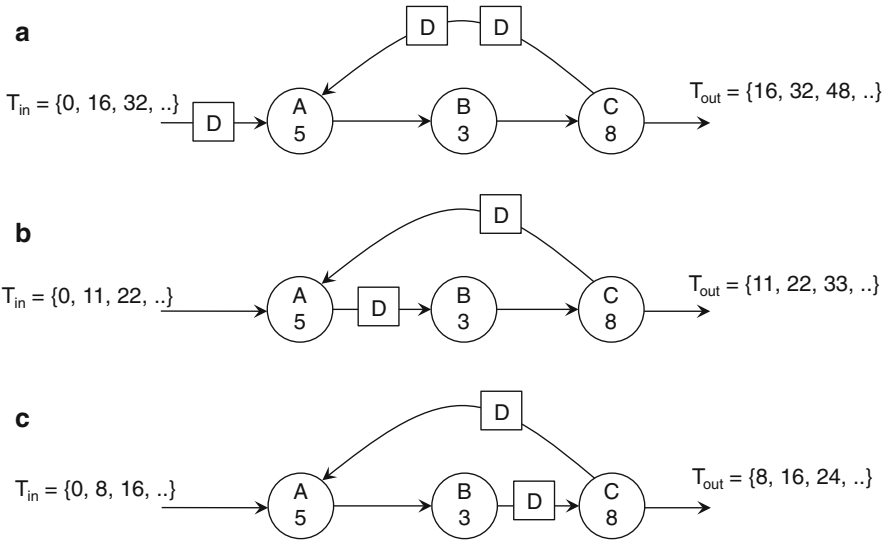


Fig. 2.21 Retiming: (**a**) Original Graph. (**b**) Graph after first re-timing transformation. (**c**) Graph after second re-timing transformation

Shifting the delay on the edge *BC* further would result in a delay on the outputs of actor *C*: one on the output queue, and one in the feedback loop. This final transformation illustrates an important property of retiming: it's not possible to increase the number of delays in a loop by means of retiming.

2.5.3 Pipelining

Pipelining increases the throughput of a data flow graph at the cost of increased latency. Pipelining can be easily understood as a combination of retiming and adding delays. Figure 2.22 demonstrates pipelining on an example. The orginal graph in Fig. 2.22a is extended with two pipeline delays in Fig. 2.22b. Adding delay stages at the input increases the latency of the graph. Before the delay stages, the system latency was 20 time units. After adding the delay stages, the system latency increases to 60 time units (3 samples with a latency of 20 time units each). The system throughput is 1 sample per 20 time units. We can now increase the system throughput by retiming the pipelined graph, so that we obtain Fig. 2.22c. The throughput of this graph is now 1 sample per 10 time units, and the latency is 30 time units (3 times 10 time units). This analysis points out an important property of pipelining: the slowest pipeline stage determines the throughput of the overall system.

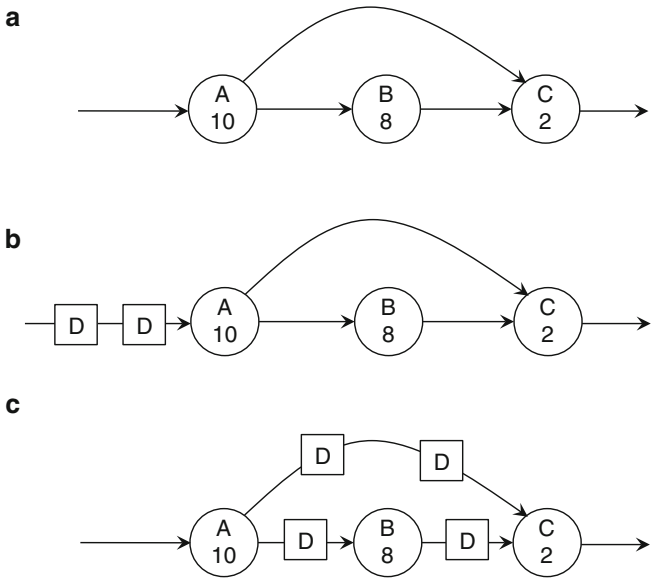


Fig. 2.22 Pipelining: (**a**) Original graph. (**b**) Graph after adding two pipeline stages. (**c**) Graph after retiming the pipeline stages

2.5.4 Unfolding

The final transformation we discuss is unfolding. The idea of unfolding is the parallel implementation of multiple instances of a given data flow graph. For example, assume a data flow graph G which processes a stream of samples. The two-unfolded graph $G2$ consists of two instances of G; this graph $G2$ processes two samples at a time.

The rules of unfolding are very similar to the rules of multi-rate expansion. Each actor A of the unfolded system is replicated the number of times needed for the unfolding. Next, the interconnections are made while respecting the sample sequence of the original system. Finally, the delays are redistributed over the interconnections.

The unfolding process is formalized as follows.

- Assume a graph G with an actor A and an edge AB carrying n delays.
- The v-unfolding of the graph G will replicate the actor A v times, namely A_0, A_1, .., A_{v-1}. The interconnection AB is replicated v times as well, AB_0, AB_1, .., AB_{v-1}.
- Edge AB_i connects A_i with B_k, for which $i : 0..v-1$ and $k = (i+n)\,\%v$.
- Edge AB_i carries $\lfloor (i+n)/v \rfloor$ delays. If $n < v$, then there will be $v-n$ edges without a delay.

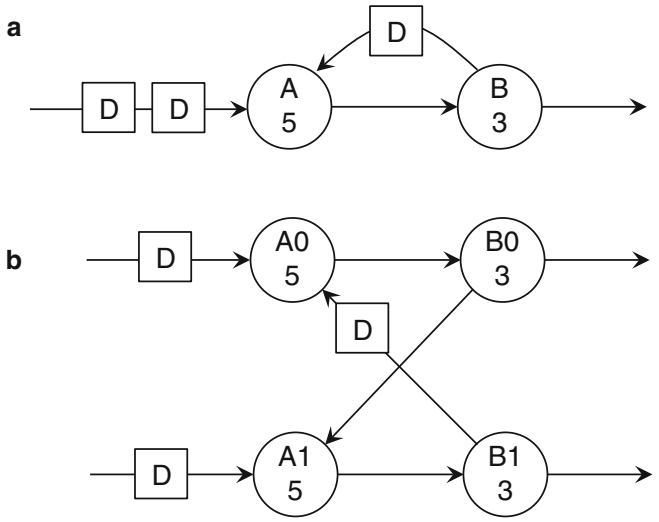


Fig. 2.23 Unfolding: (**a**) Original graph. (**b**) Graph after two-unfolding

Figure 2.23 illustrates the unfolding process with an example graph, unfolded two times. You can notice that the unfolded graph has two inputs and two outputs, and hence is able to accept twice as much data per iteration as the original data flow graph. On the other hand, unfolding the graph seems to slow it down. The critical loop now includes A_0, B_0, A_1 and B_1, but there is still only a single delay element in the overall loop. Hence, the iteration bound of a v-unfolded graph has increased v times.

Unfolding of data-flow graphs is used to process data streams with very high sample rates. In this case, the high-speed stream is expanded into v parallel streams. Stream i carries sample $s_i, s_{i+v}, s_{i+2v}, ..$ from the original stream. For example, in Fig. 2.23, the even samples would be processed by A_0 while the odd samples would be processed by A_1. However, because unfolding decreases the iteration bound, the overall computation speed of the system may be affected.

This completes our discussion on data flow graph transformations. Pipelining, retiming and unfolding are important performance-enhancing manipulations on data flow graphs, and they have a significant impact on the quality of the final implementation.

2.6 Data Flow Modeling Summary

Data flow models express concurrent systems in such as way that the models can map into hardware as well as in software. Data flow models consist of actors which communicate by means of tokens which flow over queues from one actor to the

other. A data flow model can precisely and formally express the activities of a concurrent system. An interesting class of data flow systems are synchronous data flow (SDF) models. In such models, all actors can produce or consume a fixed amount of tokens per iteration (or invocation).

By converting a given SDF graph to a topology matrix, it is possible to derive stability properties of a data flow system automatically. A stable data flow system can be executed using a periodic admissible sequential schedule (PASS), a fixed period sequence of actor firings.

Next, we introduced a resource model for data flow graphs: a high-level model to analyze the performance of a data flow graph implementation. Performance is characterized by two metrics: latency and throughput. Latency is the time it takes to compute an output sample for a corresponding input sample. Throughput is the amount of samples that can be accepted per time unit at an input or an output.

We discussed four different transformations on data flow graphs. Multi-rate expansion is useful to create a single-rate SDF from a multi-rate SDF. Retiming redistributes the delays in a data flow graph in order to improve throughput. Pipelining has the same objective, but it adds delays to a data flow graph. Unfolding creates parallel instances from a data flow graph, in order to process multiple samples from a data stream in parallel.

Data flow modeling remains an important and easy-to-understand design and modeling technique. They are very popular in signal-processing application, or any application where infinite streams of signal samples can be captured as token streams. Data flow modeling is highly relevant to hardware-software codesign because of the clear and clean manner in which it captures system specifications.

2.7 Further Reading

Data flow analysis and implementation has been well researched over the past few decades, and data flow enjoys a rich body of literature.

In the early 1970s, data flow has been considered as a replacement for traditional instruction-fetch machines. Actual data flow computers were build that operate very much according to the SDF principles discussed here. Those early years of data flow have been documented very well at a retrospective conference called *Data flow to Synthesis Retrospective*. The conference honored Arvind, one of data flows' pioneers, and the online proceedings include a talk by Jack Dennis (Dennis 2007).

In the 1980s, data flow garnered attention because of its ability to describe signal processing problems well. For example, Lee and Messerschmit described SDF scheduling mechamisms (Lee and Messerschmitt 1987). Parhi and Messerschmit discussed retiming, pipelining and unfolding transformations of SDF graphs (Parhi and Messerschmitt 1989). Lee as well as Parhi have each authored an excellent textbook that includes data flow modeling and implementation as part of the material, see (Lee and Seshia 2011) and (Parhi 1999). The work from Lee eventually gave rise to the Ptolemy environment (Eker et al. 2003). Despite these successes,

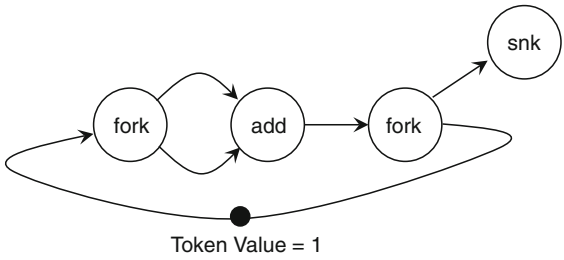


Fig. 2.24 SDF graph for Problem 2.1

data flow never became truly dominant compared to existing control-oriented paradigms. This is regrettable: a well-known, but difficult to solve, design problem that affects hardware and software designers alike is how to build parallel versions of solutions originally conceived as sequential (C) programs.

2.8 Problems

Problem 2.1. Consider the single-rate SDF graph in Fig. 2.24. The graph contains three types of actors. The fork actor reads one token and produces two copies of the input token, one on each output. The add actor adds up two tokens, producing a single token that holds the sum of the input tokens. The snk actor is a token-sink which records the sequence of tokens appearing at its input. A single initial token, with value 1, is placed in this graph. Find the value of tokens that is produced into the snk actor. Find a short-hand notation for this sequence of numbers.

Problem 2.2. The Fibonacci Number series F is defined by F(0)=0, F(1)=1, $F(i)=F(i-1)+F(i-2)$ when i is greater then 1. By changing the marking of the SDF graph in Fig. 2.27, it is possible to generate the Fibonacci series into the snk actor. Find the location and the initial value of the tokens in the modified graph.

Problem 2.3. Consider the SDF graph in Fig. 2.25. Transform that graph such that it will produce the same sequence of tokens as tuples instead of as a sequence of singletons. To implement this, replace the snk actor with snk2, an actor which requires two tokens on two different inputs in order to fire. Next make additional transformations to the graph and its marking so that it will produce this double-rate sequence into snk2.

Problem 2.4. Data Flow actors cannot contain state variables. Yet, we can 'simulate' state variables with tokens. Using only an adder actor, show how you can implement an accumulator that will obtain the sum of an infinite series of input tokens.

Problem 2.5. For the SDF graph of Fig. 2.26, find a condition between x and y for a PASS to exist.

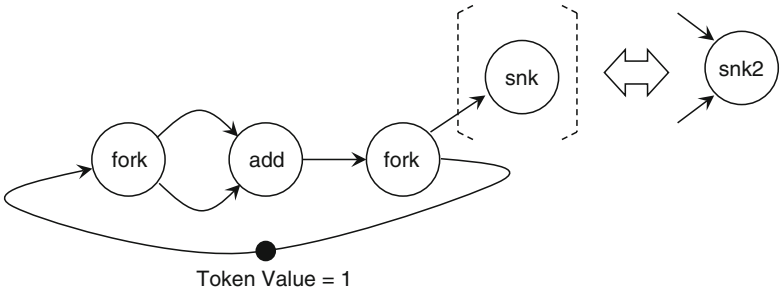


Fig. 2.25 SDF graph for Problem 2.3

Fig. 2.26 SDF graph for Problem 2.5

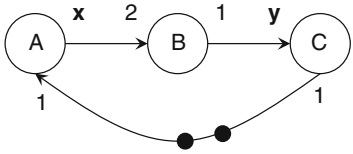


Fig. 2.27 SDF graph for Problem 2.6

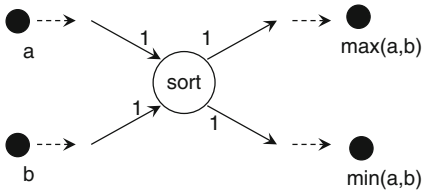
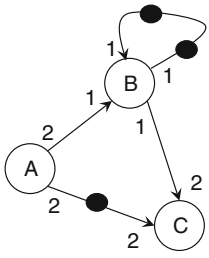


Fig. 2.28 SDF graph for Problem 2.7



Problem 2.6. Given the two-input sorting actor shown in Fig. 2.27. Using this actor, create a SDF graph of a sorting network with four inputs and four outputs.

Problem 2.7. Draw the multi-rate expansion for the multirate SDF given in Fig. 2.28. Don't forget to redistribute the initial tokens on the multirate-expanded result.

Problem 2.8. The data flow diagram in Fig. 2.29 demonstrates *reconvergent* edges: edges which go around one actor. Reconvergent edges tend to make analysis of a data flow graph a bit harder because they may imply that multiple critical loops may be laying on top of one another. This problem explores this effect.

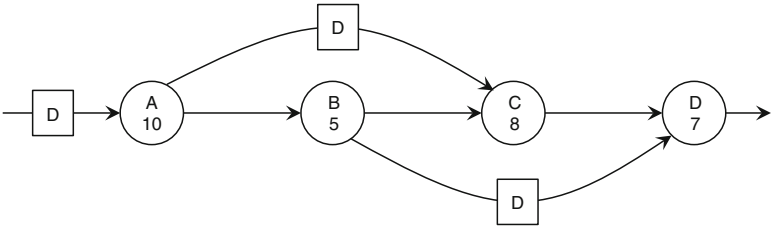


Fig. 2.29 SDF graph for Problem 2.8

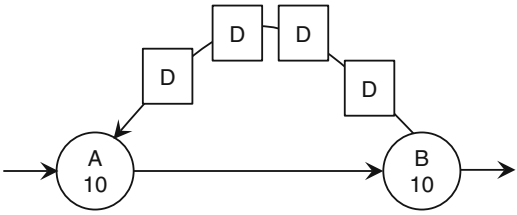


Fig. 2.30 SDF graph for Problem 2.9

(a) Determine the loop bounds of all loops in the graph of Fig. 2.29. Take the input/output constraints into account by assuming an implicit loop from output to input.
(b) Using the results of the previous part, determine the iteration bound for this graph.
(c) Find the effective throughput of this graph, based on the distribution of delays as shown in Fig. 2.29.
(d) Does the system as shown in Fig. 2.29 achieve the iteration bound? If not, apply the retiming transformation and improve the effective throughput so that you get as close as possible to the iteration bound.

Problem 2.9. Unfold the graph in Fig. 2.30 three times. Determine the iteration bound before and after the unfolding operation.

http://www.springer.com/978-1-4614-3736-9

A Practical Introduction to Hardware/Software Codesign

Schaumont, P.
2013, XXII, 482 p., Hardcover
ISBN: 978-1-4614-3736-9