Modeling Stream-Based Applications using the SBF model of computation

Bart Kienhuis and Ed F. Deprettere
Leiden University, LIACS

September, 2001

Abstract. Modeling applications and architectures at various levels of abstraction is becoming more and more an accepted approach in embedded system design. When looking at the modeling of applications in the domain of video, audio, and graphics applications, we notice that they exhibit a high degree of task parallelism and operate on streams of data. Models that we can use to specify such stream-based applications on a high level of abstraction are the dataflow models and process network models. Each of these models has its own merits. Therefore, an alternative approach is to introduce a model of computation that combines the semantics of both models of computation. In this paper, we introduce such a model of computation, which we call the *Stream-Based Functions* (SBF) model of computation and show an example. Furthermore, we discuss the composition and decomposition of SBF objects and put the SBF model of computation in the context of relevant dataflow models and process network models.

Keywords: Process Network Models, Dataflow Models, Stream-Based Functions (SBF), Composition, Decomposition

1. Introduction

Modeling applications and architectures at various levels of abstraction is becoming more and more an accepted approach in embedded system design. This has led to design methodologies in which the design space of embedded systems is narrowed down in an iterative manner. The Y-chart approach (Kienhuis et al., 1997) is an example of such a design methodology.

Video, audio, and graphics applications have in common that they exhibit a high degree of task parallelism and operate on streams of data. When modeling these applications at a high level of abstraction one wants to make the parallelism and the streaming of data explicit without imposing restrictions on the scheduling of the parallel tasks. One also wants to have the possibility to alter the degree of parallelism in the applications as it will very much impact the implementation in an embedded system. Finally, a quick feedback on the quality of potential lower level models and ultimate implementations is required. Obviously, candidate models that we can use to specify stream based applications on a high level of abstraction are dataflow models and process network models. Each of these models has its own merits. Dataflow models are appealing in that they are functional and have firing rules which make the reasoning on the models more tractable. Process networks, on the other hand, is a more general model with which task level parallelism in stream

© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

based applications can be expressed without relying on specific actors and firing rules as encountered in dataflow networks. Moreover, process network models could in principle be refined into more restricted dataflow models at lower levels of abstraction. An alternative approach is to introduce a model of computation that combines the semantics of both models of computation.

In this paper, we introduce such a model of computation. We call this model the *Stream-Based Functions* (SBF) model of computation. It is a model that is more general than any one of the determinate dataflow models. It is as general as the process network model but has process behavior that is more structured as the dataflow models are. It is a natural model for specifying stream based applications at a high level of abstraction and its structure simplifies the transition from that level to the next level down the abstraction levels found in embedded system design.

We start by introducing the SBF model of computation in section 2, which is followed by an example in section 4. In section 5, we discuss the composition and decomposition of SBF objects. In section 6, we place the SBF model of computation in the context of other relevant models of computation like dataflow and process networks and we conclude this paper in section 7.

2. Stream-Based Functions

We propose a model of computation called *Stream-Based Functions* (SBF) with which stream-based applications with a high degree of task parallelism can be naturally specified, as discussed for the first time in (Kienhuis, 1999). The essential components in this model are *Stream-Based Function Objects* and *Channels*. Stream-based applications are described as a *network* of SBF objects communicating concurrently with each other using channels. These channels interconnect SBF objects and buffer possibly unbounded streams of tokens communicated between a producing SBF object and a consuming SBF object. A network of SBF objects is a specialized *Kahn Process Network* (Kahn, 1974) in the sense that the SBF objects are structured and operate in a particular way as we will explain shortly.

An example of a process network is shown in Figure 1. It consists of a number of processes connected to each other via channels over which the processes exchange streams of tokens. The Source process produces a stream that is taken in by the filter_A process. This process produces a stream that is further processed by the filter_B process. This second filter produces two streams: one back to the filter_A process and the other to the Sink process. All processes in the network run in parallel. Synchronization between processes is by means of blocking reads. An application specification that obeys Kahn's model is determinate (Kahn, 1974; Kahn and MacQueen, 1977), which means

that its behavior is independent of the schedule of the processes in the network.

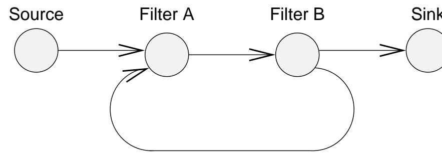

Figure 1. An example of a process network.

The sequence of statements inside a process consists of a mix of control statements, read and write statements, and function-call statements. A Kahn process does not structure these statements in any particular way. The SBF objects, on the other hand, structure these statements in a way that is reminiscent of the 'Applicative State Transition' (AST) Model described by Backus (Backus, 1978), and the related AST node in signal flow graphs proposed by Annevelink (Annevelink, 1988).

An SBF object contains three components: a *set of functions*, a *controller*, and a *state*. An enabled function consumes a number of tokens from input channels and the state, evaluates, and writes tokens to output channels and the state. By repeatedly enabling functions, an SBF object operates on streams. The controller enables the function that is associated with the current function state and determines which function it must enable next, possibly using data from the state of the SBF object.

3. The SBF Object

An SBF object has an inside view and an outside view. Inside an SBF object are present the set of functions, the controller, and the state. The set of functions is also referred to as the *function repertoire* of an SBF object. At the outside an SBF object exposes read and write *ports*. These ports connect to channels, allowing SBF objects to communicate streams with each other.

An SBF object is shown in Figure 2. The function repertoire is $P = \{f_a, f_b\}$. The two read ports and the write port connect to the unbounded FIFO buffers *Buffer0*, *Buffer1*, and *Buffer2*, respectively.

The set P of functions determines the functionality of an SBF object. P contains at least an initialization function f_{init} and is finite:

$$P = \{f_{init}, f_a, f_b, \ldots, f_x\}. \quad (1)$$

These functions evaluate within an SBF object in a *sequential order* such as,

$$f_{init}, f_a, f_b, f_a, f_b, f_a, \ldots. \quad (2)$$

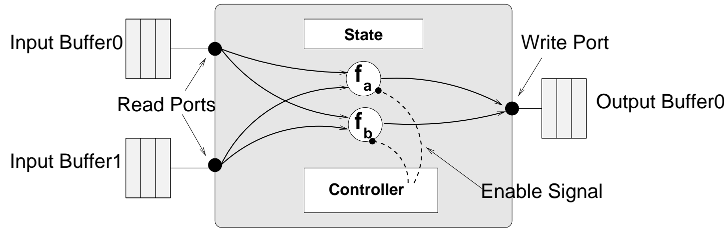

Figure 2. An SBF object.

The *controller* governs the order of function evaluations. It keeps track of the evaluation order using the variable c, called the *current function state*. Each time the current function state changes, a function *transition* takes place. The current function state belongs to the function state space C of the SBF object's state space. The data variables belong to the data state space D of the SBF object's state space. The SBF object's state space is defined as the Cartesian product of C and D,

$$S = C \times D, \;\; C \cap D = \emptyset. \tag{3}$$

3.1. FUNCTIONS

For each function f from the set P, there is a function call

$$f(x_0, \ldots, x_m, D) = (y_0, \ldots, y_n, D). \tag{4}$$

The function is capable of changing the contents of the data state space D and the input data $x_0, \ldots, x_m$ into the output data $y_0, \ldots, y_n$. A function call may have no input data or no output data, in which case it describes a *source* or a *sink* function, respectively. A source function only produces tokens and a sink function only consumes tokens. A function reads its input data $x_0, \ldots, x_m$ from read ports and writes its output data $y_0, \ldots, y_n$ to write ports. The binding of function arguments and results to read ports, write ports, and data variables is statically determined.

When the controller *enables* a function, it reads data from its input ports and the data state space, evaluates, and writes data to its output ports and the data state space. The function reads all its input data using blocking reads. State variables are immediately available. A function can read only one token at a time from the input channel that is to provide the input argument. Once a function has obtained all input data, it can not accept new data until it has written its results. During the evaluation of a function, the state S cannot change and does not change. Consequently, a function operates without any side effects. When the enabled function has written all its results, we say the function has *fired*. After firing, the current function informs the controller

that the next function state can be computed, hence the next function can be enabled.

The current function reads its arguments one at a time and in a prescribed order using a blocking read. This prohibits the testing of a read port on the availability of tokens. As a consequence, a function cannot have another behavior based on the results of testing a condition; thus it evaluates unconditionally, which gives the SBF object its deterministic behavior. A function writes the output data $y_0, \ldots, y_n$ to the appropriate write port using a non-blocking write, also in a prescribed linear order.

3.2. CONTROLLER

The controller keeps track of the function invocations using the function state variable c. It moves from the current function state c to the next function state c' whenever the current function has fired. The controller has a *transition function* ω, and a *binding function* μ. The transition function is a map from $C \times D$ to C,

$$\omega : C \times D \to C, \;\; \omega(c, d) = c'. \tag{5}$$

To determine the next function state c', the transition function observes the function state space C and the data state space D. The controller cannot change the content of D; it can only observe it. Although the controller is not connected to any read or write ports, the transition function describes dynamic behavior. If both C and D are observed to determine the next function state, the function state can traverse an infinite number of trajectories. In a more restricted case, when the transition function observes only C to determine the next function state, it always describes a single path in C. Furthermore, if this path is finite, the transition function determines a *cycle*, corresponding to a cyclo-static schedule (Bilsen et al., 1995).

At each state, a specific function needs to be evaluated as determined by the *binding function* μ. This binding function associates a function f from the set P with a particular function state c. Only one function can be associated with a particular function state.

$$\mu : C \to P, \;\; \mu(c) = f \tag{6}$$

Using the transition function ω and the binding function μ, the controller repeatedly invokes and enables a function from the set P of functions, thereby generating a stream of functions as follows:

$$f_{init} \stackrel{\mu(\omega(S))}{\longrightarrow} f_a \stackrel{\mu(\omega(S))}{\longrightarrow} f_b \stackrel{\mu(\omega(S))}{\longrightarrow} \ldots f_x \stackrel{\mu(\omega(S))}{\longrightarrow} \ldots \tag{7}$$

The evaluation of the ω and μ functions takes place instantaneously. Thus an SBF object operates on streams by successively enabling a function from the

set of functions P that reads from input ports and writes to output ports. This behavior is often referred to as a *Fire-and-Exit* behavior.

When an SBF object is created, the controller needs to start at a particular function state c. A special *initialization* function f_{init} is available in the function repertoire P that initializes the function state c. To that end, this function evaluates first and only once. To determine the initialization value of c, the initialization function may read tokens from one or more input ports.

4. Example of an SBF Object operation

We illustrate the operation of an SBF object using the object shown in Figure 3. Its function repertoire is $\{\mathsf{f_a}, \mathsf{f_b}, \mathsf{f_c}\}$. For the sake of brevity, we have left out the initialization function f_{init}. There are two state variables, the function state variable c, an element of C, and the data state variable d, an element of D.

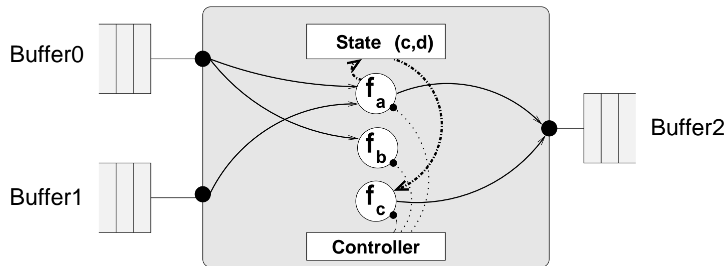

Figure 3. An SBF object with function repertoire $\{\mathsf{f_a}, \mathsf{f_b}, \mathsf{f_c}\}$, and state variables c and d.

Function $\mathsf{f_a}$ reads input data from the two read ports and writes output data to the write port. It also writes to D. Function $\mathsf{f_b}$ reads only input data from the read port connected to Buffer0. Function f_c reads from D, and writes output data to the write port.

If function $\mathsf{f_a}$ is enabled, it reads first one token from Buffer0 and then one token from Buffer1. If Buffer0 does not contain any tokens, the complete SBF object blocks until a token becomes available in Buffer0 even though data might already be available on Buffer1. When both tokens have been read, the function $\mathsf{f_a}$ evaluates. The resulting token is written to Buffer2. The function also writes d to D. This terminates the firing of $\mathsf{f_a}$ and the controller enables the next function, which is $\mathsf{f_b}$. This function reads a token from Buffer0 and evaluates. A transition takes place and the next function enabled is again function $\mathsf{f_a}$. After this function has fired, the controller enables function f_c. This function reads d from D and immediately evaluates because it requires no input from the buffers. It writes the resulting token to Buffer2. Again a transition takes place and the next function enabled is

function f_a, followed by f_b, f_a, and f_c. Because the controller describes a cyclic sequence, the SBF object's behavior is cyclic.

In the example, the controller enables the functions in the sequence f_a, f_b, f_a, f_c. This sequence is obtained with the binding function

$$\mu(c) = \begin{cases} f_a, & \text{if } c = 0 \\ f_b, & \text{if } c = 1 \\ f_a, & \text{if } c = 2 \\ f_c, & \text{if } c = 3, \end{cases} \quad (8)$$

and the transition function

$$\omega(c) = c + 1 \pmod 3. \quad (9)$$

The sequence of functions results in a particular consumption/production pattern of tokens. These patterns are shown in Table I for the SBF object in our example. It shows the four function states, the functions executed at these states, the data state variable d and the three buffers from which the functions read tokens (R) or to which they write tokens (W).

Table I. Token consumption/production pattern of the SBF object shown in Figure 3.

Current Function State	Function	Data state variable d	Buffer0	Buffer1	Buffer2
c_0	f_a	W	R	R	W
c_1	f_b		R		
c_2	f_a	W	R	R	W
c_3	f_c	R			W

The SBF object shown in Figure 3 could, for example, implement the filter_A process given in Figure 1. In that case, Buffer0 implements the channel between the processes Sink and filter_A. Buffer1 implements the feedback channel between the processes filter_B and filter_A. Finally, Buffer2 implements the channel between the processes filter_A and filter_B.

4.1. Deadlock

The SBF objects in a network run autonomously and schedule themselves on the availability of data on the channels. On the other hand, the functions inside an SBF object are scheduled by means of the transition function. When scheduling these functions inside an SBF object, we should make sure that

we do not introduce *deadlock*. In Figure 4 we show two SBF objects, SBF_0 and SBF_1, connected to each other via three buffers, Buffer0, Buffer1, and Buffer2. SBF_0 is scheduled in such a way that it first produces a token on Buffer0 and then reads a token from Buffer1 and, finally, produces a token on Buffer2. The numbers 1 to 3 from top to bottom indicates this sequence. The other SBF object, SBF_1, tries to read a token from Buffer2 first, then produces a token on Buffer1 and, finally, reads a token from Buffer0. This sequence is indicated by the numbers 1 to 3 from bottom to top.

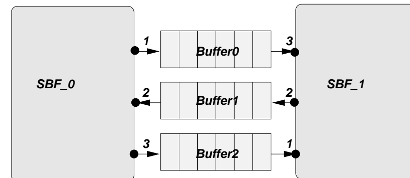

Figure 4. Scheduling functions inside an SBF object may lead to deadlock.

The situation presented in Figure 4 will deadlock. SBF_0 is able to produce a token on Buffer0, but blocks when trying to read a token from Buffer1. On the other side, SBF_1 tries to read a token from Buffer2, but blocks because no token is ever produced on this buffer because SBF_0 blocks while reading from Buffer2. Thus, if two autonomous SBF objects are communicating, care should be taken in scheduling the functions inside the SBF objects to prevent deadlock from occurring.

5. Networks of SBF Objects

We specify stream-based applications as networks of SBF objects. The SBF objects in the network run in parallel whereas inside each SBF object a sequential invocation of functions takes place. We can decrease the amount of parallelism in an application by combining SBF objects. Similarly, we can increase the amount of parallelism in an application by decomposing an SBF object. To do so, we have to consider composition and decomposition of SBF objects. In general, composition/decomposition of process networks is not trivial as the functions and control are intertwined. We believe that composing and decomposing SBF objects is less cumbersome due to the structure of SBF objects.

To explain composition and decomposition of SBF objects, we have to introduce the notion of *function variants*; A function in an SBF object is bound statically to input ports, output ports and data state variables. A *function variant* f' of function f has the same functional behavior as f but is bound to different ports and data state variables. The repertoire of an SBF object can contain both functions and function variants.

5.1. Composition of SBF Objects

The construction of an SBF object that is a composition of two SBF objects, takes the following steps:

1. Combine the two sets of functions P and Q of both SBF objects to obtain the new set $Z = P \cup Q$. The set Z will contain variants of functions, if necessary.

2. Combine the states S and S' of both SBF objects to get the new state $W = S + S'$.

3. If one or more channels between the original two SBF objects are include by the new SBF object, make the channels self-loops of the new SBF object.

4. Construct a new schedule that sequentially interleaves the functions of set Z to determine a new transition function ω and a new binding function μ for the new SBF object.

As an example, consider Figure 5 in which two cases are shown, i.e., (1) and (2). Two SBF objects A and B are combined, as indicated by the enclosing box, to form a single SBF object. SBF object A contains the set of functions $P = \{f_a, f_b\}$ and the state variable x. SBF object B contains the set of functions $Q = \{f_a, f_c\}$ and state variable y. Function state variables are not shown. Both SBF objects have only one read port and one write port.

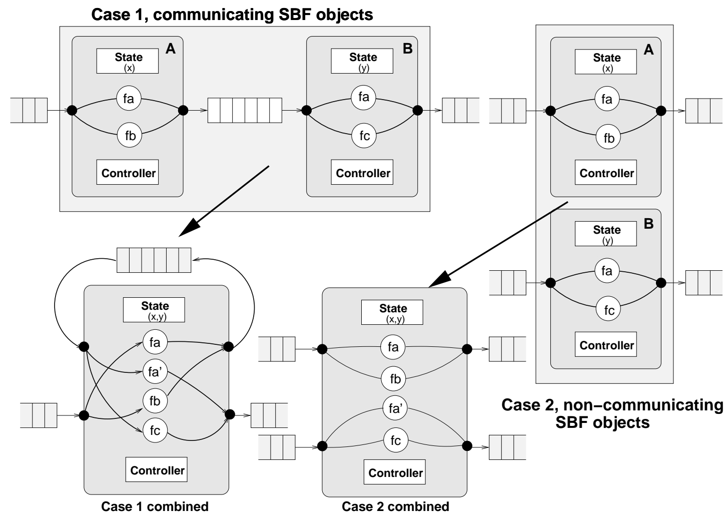

Figure 5. Two cases of composition of SBF objects.

In case 1, we combine two SBF objects that communicate with each other over a channel. As a consequence, the channel is enclosed by the new combined SBF object. According to the four steps given above, we first combine the functions of sets P and Q to form the new set $Z = \{f_a, f'_a, f_b, f_c\}$, in which the function f'_a is a variant of the function f_a. Secondly, we combine the state of both SBF objects to obtain the new state W containing variables x and y. Thirdly, because the new SBF object encloses the communication channel, we make the channel a self-loop of the new SBF object. As a consequence, the functions f_a and f_b write to the self-loop channel, and the functions f'_a and f_c read from the self-loop channel. Fourthly, and finally, we schedule the functions in function repertoire Z.

In case 2, we combine two non-communicating SBF objects into one new SBF object. According to the four composition steps, we first combine the functions of sets P and Q to form the new set $Z = \{f_a, f'_a, f_b, f_c\}$. Secondly, we combine the state of both SBF objects to obtain the new state W containing the variables x and y. We can omit the third step because no channel is enclosed by the final SBF object. Fourthly, and finally, we schedule the functions in the function set Z. Where the SBF objects A and B operate in parallel, the new SBF object executes the functions sequentially. Therefore, in the fourth step, we have to interleave the functions such that they execute, for example, one after the other leading to a reduction in parallelism.

In Figure 5(1), we combine two SBF objects that enclose a channel into a new SBF object. According to step 3, we make the enclosed channel a self-loop of the new SBF object, a behavior preserving step. Yet, this self-loop has become part of the SBF object in which the functions are scheduled and will only be accessed by functions in the repertoire; not by some other SBF object. As a consequence, if we are able to find a static, possibly cyclic, schedule for the functions in the SBF object, we can find an upperbound on the buffer size and install a buffer of this size in the data space D of the new SBF object. This way, we can absorb unbounded FIFOs representing self-loops into SBF objects.

We have to construct a new schedule of the repertoire functions for the new combined SBF object, as described by the fourth step. In general, finding such schedule is hard as it can easily lead to deadlock as discussed in Section 4.1 and requires global knowledge about the network. Nonetheless, for some restrictive cases, it is possible to obtain a sequential schedule of the repertoire functions at compile time. We use the SBF model in the *Compaan* compiler project (Kienhuis et al., 2000), which automatically transforms nested loop programs described in Matlab into networks of SBF objects. A Matlab program is evaluated correctly in a sequential way and we exploit this schedule to construct at compile time a valid schedule for the repertoire functions of the combined SBF object.

5.2. Decomposition of SBF Objects

Decomposing an SBF object is a more difficult process than SBF composition. When decomposing an SBF object, we need to determine the available parallelism in the object, which means finding parallelism between the functions of the repertoire. This leads to a data-dependence analysis, and in general, a data-dependence analysis is known to be a hard problem, especially when dynamic behavior is involved (Banerjee, 1988). However, in case the schedule of invocation of the repertoire functions can be described as a nested-loop program, a complete static analysis is possible (Held, 1996). Based on this analysis, and within the context of *Compaan*, we have found a set of transformations like unfolding, skewing, and plane cutting, to automatically decompose SBF objects (Stefanov et al., 2002).

Each function in the repertoire can be described by itself in a sequential formalism (e.g., the C language). Again, using the data-dependence analysis technique discussed in (Held, 1996), the function can be decomposed in a new set of repertoire functions that are more primitive. Using this new set of repertoire functions, new decompositions of the SBF object can result. In principle, the creation of new repertoire functions and decomposition can continue until the set of functions consists of only a single atomic function. SBF objects in this case have a function state space equal to $C = \{c\}$ and the transition function uniquely gives $c = 1$. The binding function binds the single function to this single state.

6. Related Work

Many models of computation have been proposed over the years to describe stream-based applications. The models that are relevant to the proposed SBF model are the dataflow, process, and mixed models.

6.1. Dataflow Models

The dataflow model of computation describes stream-based applications in a natural way and makes function parallelism explicit. It describes applications as a network of dataflow *actors* that perform a particular computation. Actors connect with each other via buffers, allowing them to communicate tokens with each other. When an actor *fires*, it consumes tokens, evaluates, and produces new tokens. The condition under which an actor is able to fire, is determined by a one or more *firing rules*. These rules are checked by a global *scheduler*. Therefore, the scheduler determines the ordering of the actors at either compile time or at run-time. The most well-known dataflow models are homogeneous dataflow (HDF) (Veen, 1986), synchronous dataflow

(SDF) (Lee and Messerschmitt, 1987), cyclo-static dataflow (CSDF) (Bilsen et al., 1995) and dynamic dataflow (DDF) (Jagannathan, 1995).

The SBF model can act as dataflow models HDF, SDF, and CSDF. In the case of HDF and SDF, the set of functions contains only one function. The transition function is equal to $c = 1$, to which the binding function binds the only function present. In case of SDF, multi-rate behavior is captured as an uninterrupted sequence of reads and writes of single tokens. In case of CSDF, the set P contains more than one function and the transition function traverses a cycle by observing only the function state space C to determine the next state, leading to the mapping $C \rightarrow C$.

The SBF model cannot act as DDF since this model is non-deterministic. However, it can act as deterministic DDF as described, for example, in (Buck, 1993). As a consequence, the SBF model cannot describe the classic example of the non-deterministic merge (Lee and Parks, 1995). But it is capable of describing *data-dependent* behavior, for example, a variable length decoder. In case of deterministic DDF, the set P contains one or more functions and the transition function determines the next state using the general mapping $C \times D \rightarrow C$.

6.2. Process Models

Process models can also be used to describe stream-based applications. Process models describe an application as a network of processes communicating with each other via buffers. Different from dataflow models, a process proceeds autonomously, i.e., it is not controlled by a global scheduler. A process interacts according to a particular *protocol* with other processes in a network. Two well-known process models are *Kahn Process Networks* (Kahn, 1974; Kahn and MacQueen, 1977) and *Communicating Sequential Processes* (Hoare, 1978; Hoare, 1985).

Kahn processes run forever and use unbounded FIFOs as buffers. To synchronize onto these buffers, processes use a blocking read protocol. As a consequence, a Kahn process cannot poll the read ports and hence describes deterministic behavior. In contrast, CSP processes typically terminate and use single place buffers. The exchange of data between two processes happens using a *rendezvous* protocol. In CSP, a process can choose from which port to read first and hence is able to describes non-deterministic behavior.

6.3. Combined Dataflow/Process Models

A combination of dataflow models and process models already exist and examples are the *Dataflow Process Network* model and the *Application State Transition* model. The former model combines the Kahn semantics with dataflow, whereas the latter model combines a CSP kind of semantics with dataflow.

In the dataflow process network model (DPN) (Lee and Parks, 1995), a single process describes a dataflow actor and a set of *sequential firing rules*. The process itself, instead of a global scheduler, checks which firing rule applies. It does this in a sequential order using blocking reads, referred to as *sequential firing rules*. If a valid firing rule is found, the process enables the actor. At that moment, all input arguments of the function must be present and the actor evaluates instantaneously. Tokens are written to the output ports using a non-blocking write. After evaluating, the process checks again the firing rules until a valid firing of the actor is found.

In the Application State Transition Model (AST) (Annevelink, 1988), a single process contains a set of functions and state, whereby each function reads input data from specific read ports and/or state variables and writes data to specific write ports and/or state variables. The functions communicate with each other using single token passing in a rendezvous kind of style, very similar to CSP. When one function from the set is active, other functions are idle until it has evaluated. Within the AST mode, an AST node has a special control port, from which the special control function reads tokens to determine the function to evaluate next. An AST process is more restrictive than a CSP process, since the use of non-deterministic process constructs is not possible. Because this model was inspired by the concept of the applicative state transition model presented by (Backus, 1978), it is named accordingly.

The SBF model differs from the DPN model in that the functions themselves check for the availability of data whereas in the DPN model, data is available when a function is enabled. As a consequence, an SBF object is closer to a possible hardware implementation. The SBF and DPN model both describe deterministic DDF.

The SBF model differs from the AST model in that the communication is based on Kahn semantics whereas in the AST model it is based on CSP kind of semantics. For stream-based applications, the Kahn semantics is more appropriated. In the AST model there is not controller present. Instead a control port is used on which a sequence of tokens is present that will invocate the functions in the AST node in a particular order. Because of the single token communication model, the AST model describes the class of HDF and is therefore less general than the SBF model.

7. Conclusions

In this paper we have presented the SBF model of computation that is a combination of dataflow models and process network models. It combines the strong characteristic of dataflow, i.e., being functional and structured, with the strong characteristics of process networks, i.e., schedule freedom

and deterministic behavior. As a result, the SBF model of computation is well suited to describe stream-based applications.

Applications specified as process networks or as a network of SBF objects, are equivalent, e.g., they can both model the same class of applications. One can transform a Kahn process network into an SBF object and vice versa. The big difference between the two kinds of specifications is that the SBF objects have structure in terms of the controller, the state, and the set of functions.

The structure of an SBF object has a close resemblance with models of architectures at lower abstraction levels. However, the structure of an SBF object does only hint at hardware implementations. It can also be used for implementations in software. The structure can also be exploited to compose and decompose SBF objects to change the degree of parallelism in an application description. In case of a composition, an algorithm can be given on how to combine two or more SBF objects. A decomposition is however less trivial. Being able to compose and decompose SBF objects is important as the degree of parallelism has a large impact on the implementation of the application.

We use the SBF model in the *Compaan* compiler project (Kienhuis et al., 2000). This compiler automatically transforms nested loop programs described in Matlab into networks of SBF objects. Composition and decomposition is currently being research for inclusion in the Compaan compiler.

Finally, we remark that to simulate a network of SBF objects, we have developed the fast simulator *SBFsim* in C++, based on a simple multithreading package (Kienhuis, 1999). Alternatively, we use the process network (PN) domain in the *Ptolemy II* environment to simulate a network of SBF objects (Kienhuis et al., 2000). In these simulators, bounded FIFOs are used and the procedure described in (Parks, 1995) is applied to dynamically resize FIFOs to avoid deadlock.

Acknowledgements

This research has been performed at both Philips Research and Delft University of Technology, as part of their “cluster program”. Philips Research, Ministry of Economic affairs, and Delft University of Technology have supported this research and are hereby acknowledged.

References

Annevelink, J.: 1988, ‘HiFi, A Design Method for Implementing Signal Processing Algorithms on VLSI Processor Arrays’. Ph.D. thesis, Delft University of Technology.

Backus, J.: 1978, ‘Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs’. *Communications of the ACM* **21**(8), 613 – 641.

Banerjee, U.: 1988, *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers.

Bilsen, G., M. Engels, R. Lauwereins, and J. Peperstraete: 1995, 'Cyclo-Static Data Flow'. In: *IEEE International Conference ASSP*. pp. 3255 – 3258.

Buck, J.: 1993, 'Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model'. Ph.D. thesis, Dept. of EECS, University of California at Berkeley. Tech. Report UCB/ERL 93/69.

Held, P.: 1996, 'Functional Design of Dataflow Networks'. Ph.D. thesis, Delft University of Technology.

Hoare, C.: 1978, 'Communicating Sequential Processes'. *Communications of the ACM* **21**(8), 666 – 677.

Hoare, C.: 1985, *Communicating Sequential Processes*. Prentice-Hall.

Jagannathan, R.: 1995, 'Dataflow Models'. In: E. Zomaya (ed.): *Parallel and Distributed Computing Handbook*. McGraw-Hill.

Kahn, G.: 1974, 'The Semantics of a Simple Language For Parallel Programming'. In: *Proc. of the IFIP Congress 74*. North-Holland Publishing Co.

Kahn, G. and D. B. MacQueen: 1977, 'Coroutines and Networks of Parallel Processes'. In: *Proc. of the IFIP Congress 77*. pp. 993 – 998, North-Holland Publishing Company Co.

Kienhuis, A.: 1999, 'Design Space Exploration of Stream-based Dataflow Architectures: Method and Tools'. Ph.D. thesis, Delft University of Technology.

Kienhuis, B., E. Deprettere, K. Vissers, and P. van der Wolf: 1997, 'An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures'. In: *Proceedings of 11th Int. Conference of Applications-specific Systems, Architectures and Processors (ASAP'97)*. Zurich, Switzerland, pp. 338 – 349.

Kienhuis, B., E. Rijpkema, and E. F. Deprettere: 2000, 'Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures'. In: *8th International Workshop on Hardware/Software Codesign (CODES'2000)*. San Diego, USA.

Lee, E. A. and D. G. Messerschmitt: 1987, 'Synchronous Data flow'. *Proc. IEEE* **75**(9), 1235 – 1245.

Lee, E. A. and T. M. Parks: 1995, 'Dataflow Process Networks'. *Proceedings of the IEEE* **83**(5), 773–799.

Parks, T.: 1995, 'Bounded Scheduling of Process Networks'. Ph.D. thesis, University of California at Berkeley.

Stefanov, T., B. Kienhuis, and E. Deprettere: 2002, 'Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances'. In: *Proceedings of 10th International Symposium on Hardware/Software Codesign*. Colorado, USA.

Veen, A. H.: 1986, 'Dataflow Machine Architecture'. *ACM Computing Surveys* **18**(4), 366–396.