Dynamic Dataflow Graphs

Shuvra S. Bhattacharyya, Ed F. Deprettere, and Bart D. Theelen

Abstract Much of the work to date on dataflow models for signal processing system design has focused on decidable dataflow models that are best suited for one-dimensional signal processing. This chapter reviews more general dataflow modeling techniques that are targeted to applications that include multidimensional signal processing and dynamic dataflow behavior. As dataflow techniques are applied to signal processing systems that are more complex, and demand increasing degrees of agility and flexibility, these classes of more general dataflow models are of correspondingly increasing interest. We first provide a motivation for dynamic dataflow models of computation, and review a number of specific methods that have emerged in this class of models. Our coverage of dynamic dataflow models in this chapter includes Boolean dataflow, CAL, parameterized dataflow, enable-invoke dataflow, dynamic polyhedral process networks, scenario aware dataflow, and a stream-based function actor model.

1 Motivation for Dynamic DSP-Oriented Dataflow Models

The decidable dataflow models covered in [31] are useful for their predictability, strong formal properties, and amenability to powerful optimization techniques. However, for many signal processing applications, it is not possible to represent

S.S. Bhattacharyya (✉)
University of Maryland, College Park, MD, USA
e-mail: ssb@umd.edu

E.F. Deprettere
Leiden University, Leiden, The Netherlands
e-mail: edd@liacs.nl

B.D. Theelen
Embedded Systems Innovation by TNO, Eindhoven, The Netherlands
e-mail: bart.theelen@tno.nl

all of the functionality in terms of purely decidable dataflow representations. For example, functionality that involves conditional execution of dataflow subsystems or actors with dynamically varying production and consumption rates generally cannot be expressed in decidable dataflow models.

The need for expressive power beyond that provided by decidable dataflow techniques is becoming increasingly important in design and implementation signal processing systems. This is due to the increasing levels of application dynamics that must be supported in such systems, such as the need to support multi-standard and other forms of multi-mode signal processing operation; variable data rate processing; and complex forms of adaptive signal processing behaviors.

Intuitively, *dynamic dataflow* models can be viewed as dataflow modeling techniques in which the production and consumption rates of actors can vary in ways that are not entirely predictable at compile time. It is possible to define dynamic dataflow modeling formats that are decidable. For example, by restricting the types of dynamic dataflow actors, and by restricting the usage of such actors to a small set of graph patterns or "schemas", Gao, Govindarajan, and Panangaden defined the class of *well-behaved dataflow graphs*, which provides a dynamic dataflow modeling environment that is amenable to compile-time bounded memory verification [19].

However, most existing DSP-oriented dynamic dataflow modeling techniques do not provide decidable dataflow modeling capabilities. In other words, in exchange for the increased modeling flexibility (expressive power) provided by such techniques, one must typically give up guarantees on compile-time buffer underflow (deadlock) and overflow validation. In dynamic dataflow environments, analysis techniques may succeed in guaranteeing avoidance of buffer underflow and overflow for a significant subset of specifications, but, in general, specifications may arise that "break" these analysis techniques—i.e., that result in inconclusive results from compile-time analysis.

Dynamic dataflow techniques can be divided into two general classes: (1) those that are formulated explicitly in terms of interacting combinations of state machines and dataflow graphs, where the dataflow dynamics are represented directly in terms of transitions within one or more underlying state machines; and (2) those where the dataflow dynamics are represented using alternative means. The separation in this dichotomy can become somewhat blurry for models that have a well-defined state structure governing the dataflow dynamics, but whose design interface does not expose this structure directly to the programmer. Dynamic dataflow techniques in the first category described above are covered in [31]—in particular, those based on explicit interactions between dataflow graphs and finite state machines. This chapter focusses on the second category.[1] Specifically, dynamic dataflow modeling techniques that involve different kinds of modeling abstractions, apart from state transitions, as the key mechanisms for capturing dataflow behaviors and their potential for run-time variation.

[1]Except for the Scenario Aware Dataflow model in Sect. 6.

Numerous dynamic dataflow modeling techniques have evolved over the past couple of decades. A comprehensive coverage of these techniques, even after excluding the "state-centric" ones, is out of the scope this chapter. The objective is to provide a representative cross-section of relevant dynamic dataflow techniques, with emphasis on techniques for which useful forms of compile time analysis have been developed. Such techniques can be important for exploiting the specialized properties exposed by these models, and improving predictability and efficiency when deriving simulations or implementations.

2 Boolean Dataflow

The Boolean dataflow (BDF) model of computation extends synchronous dataflow with a class of dynamic dataflow actors in which production and consumption rates on actor ports can vary as two-valued functions of *control tokens*, which are consumed from or produced onto designated *control ports* of dynamic dataflow actors. An actor input port is referred to as a *conditional input port* if its consumption rate can vary in such a way, and similarly an output port with a dynamically varying production rate under this model is referred to as a *conditional output port*.

Given a conditional input port p of a BDF actor A, there is a corresponding input port C_p, called the *control input* for p, such that the consumption rate on C_p is statically fixed at one token per invocation of A, and the number of tokens consumed from p during a given invocation of A is a two-valued function of the data value that is consumed from C_p during the same invocation.

The dynamic dataflow behavior for a conditional output port is characterized in a similar way, except that the number of tokens produced on such a port can be a two-valued function of a token that is consumed from a control input port or of a token that is produced onto a *control output* port. If a conditional output port q is controlled by a control output port C_q, then the production rate on the control output is statically fixed at one token per actor invocation, and the number of tokens produced on q during a given invocation of the enclosing actor is a two-valued function of the data value that is produced onto C_q during the same invocation of the actor.

Two fundamental dynamic dataflow actors in BDF are the *switch* and *select* actors, which are illustrated in Fig. 1a. The switch actor has two input ports, a control input port w_c and a *data* input port w_d, and two output ports w_x and w_y. The port w_c accepts Boolean valued tokens, and the consumption rate on w_d is statically fixed at one token per actor invocation. On a given invocation of a switch actor, the data value consumed from w_d is copied to a token that is produced on either w_x or w_y depending on the Boolean value consumed from w_c. If this Boolean value is `true`, then the value from the data input is routed to w_x, and no token is produced on w_y. Conversely if the control token value is `false`, then the value from w_d is routed to w_y with no token produced on w_x.

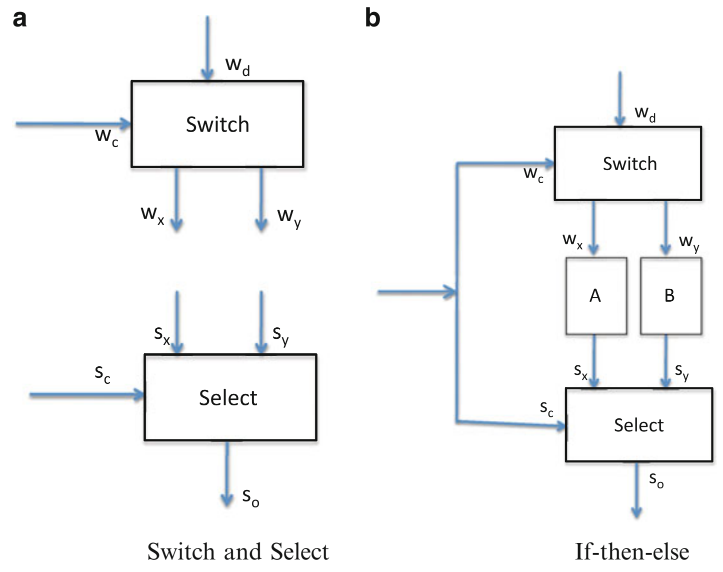


Fig. 1 (**a**) Switch and select actors in Boolean dataflow, and (**b**) an if-then-else construct expressed in terms of Boolean dataflow

A BDF select actor has a single control input port s_c; two additional input ports (*data input ports*) s_x and s_y; and a single output port s_o. Similar to the control port of the switch actor, the s_c port accepts Boolean valued tokens, and the production rate on the s_o port is statically fixed at one token per invocation. On each invocation of the select actor data is copied from a single token from either s_x or s_y to s_o depending on whether the corresponding control token value is `true` or `false` respectively.

Switch and select actors can be integrated along with other actors in various ways to express different kinds of control constructs. For example, Fig. 1b illustrates an if-then-else construct, where the actors A and B are applied conditionally based on a stream of control tokens. Here A and B are synchronous dataflow (SDF) actors that each consume one token and produce one token on each invocation.

Buck has developed scheduling techniques to automatically derive efficient control structures from BDF graphs under certain conditions [11]. Buck has also shown that BDF is Turing complete, and furthermore, that SDF augmented with just switch and select (and no other dynamic dataflow actors) is also Turing complete. This latter result provides a convenient framework with which one can demonstrate Turing completeness for other kinds of dynamic dataflow models, such as the enable-invoke dataflow model described in Sect. 5. In particular, if a given model of computation can express all SDF actors as well as the functionality associated with the BDF switch and select actors, then such a model can be shown to be Turing complete.

3 CAL

In addition to providing a dynamic dataflow model of computation that is suitable for signal processing system design, CAL provides a complete programming language and is supported by a growing family of development tools for hardware and software implementation. The name "CAL" is derived as a self-referential acronym for the *CAL actor language*. CAL was developed by Eker and Janneck at U.C. Berkeley [14], and has since evolved into an actively-developed, widely-investigated language for design and implementation of embedded software and field programmable gate array applications (e.g., see [30, 56, 77]). One of the most notable developments to date in the evolution of CAL has been its adoption as part of the recent MPEG standard for reconfigurable video coding (RVC) [8].

A CAL program is specified as a network of CAL actors, where each actor is a dataflow component that is expressed in terms of a general underlying form of dataflow. This general form of dataflow admits both static and dynamic behaviors, and even non-deterministic behaviors.

Like typical actors in any dataflow programming environment, a CAL actor in general has a set of input ports and a set of output ports that define interfaces to the enclosing dataflow graph. A CAL actor also encapsulates its own private state, which can be modified by the actor as it executes but cannot be modified directly by other actors.

The functional specification of a CAL actor is decomposed into a set of *actions*, where each action can be viewed as a template for a specific class of firings or invocations of the actor. Each firing of an actor corresponds to a specific action and executes based on the code that is associated with that action. The core functionality of actors therefore is embedded within the code of the actions. Actions can in general consume tokens from actor input ports, produce tokens on output ports, modify the actor state, and perform computation in terms of the actor state and the data obtained from any consumed tokens.

The number of tokens produced and consumed by each action with respect to each actor output and input port, respectively, is declared up front as part of the declaration of the action. An action need not consume data from all input ports nor must it produce data on all output ports, but ports with which the action exchanges data, and the associated rates of production and consumption must be constant for the action. Across different actions, however, there is no restriction of uniformity in production and consumption rates, and this flexibility enables the modeling of dynamic dataflow in CAL.

A CAL actor A can be represented as a sequence of four elements

$$\sigma_0(A), \Sigma(A), \Gamma(A), pri(A), \tag{1}$$

where $\Sigma(A)$ represents the set of all possible values that the state of A can take on; $\sigma_0(A) \in \Sigma(A)$ represents the initial state of the actor, before any actor in the enclosing dataflow graph has started execution; $\Gamma(A)$ represents the set of actions of A; and $pri(A)$ is a partial order relation, called the *priority relation* of A, on $\Gamma(A)$ that specifies relative priorities between actions.

Actions execute based on associated *guard conditions* as well as the priority relation of the enclosing actor. More specifically, each actor has an associated guard condition, which can be viewed as a Boolean expression in terms of the values of actor input tokens and actor state. An actor A can execute whenever its associated guard condition is satisfied (*true*-valued), and no higher-priority action (based on the priority relation $pri(A)$) has a guard condition that is also satisfied.

In summary, CAL is a language for describing dataflow actors in terms of ports, actions (firing templates), guards, priorities, and state. This finer, *intra-actor* granularity of formal modeling within CAL allows for novel forms of automated analysis for extracting restricted forms of dataflow structure. Such restricted forms of structure can be exploited with specialized techniques for verification or synthesis to derive more predictable or efficient implementations.

An example of this capability for *specialized region detection* in CAL programs is the technique of deriving and exploiting so-called *statically schedulable regions* (SSRs) [30]. Intuitively, an SSR is a collection of CAL actions and ports that can be scheduled and optimized statically using the full power of static dataflow techniques, such as those available for SDF, and integrated into the schedule for the overall CAL program through a top-level dynamic scheduling interface.

SSRs can be derived through a series of transformations that are applied on intermediate graph representations. These representations capture detailed relationships among actor ports and actions, and provide a framework for effective *quasi-static scheduling* of CAL-based dynamic dataflow representations. Quasi-static scheduling is the construction of dataflow graph schedules in which a significant proportion of overall schedule structure is fixed at compile-time. Quasi-static scheduling has the potential to significantly improve predictability, reduce run-time scheduling overhead, and as discussed above, expose subsystems whose internal schedules can be generated using purely static dataflow scheduling techniques.

Further discussion about CAL can be found in [42], which discusses the application of CAL to reconfigurable video coding.

4 Parameterized Dataflow

Parameterized dataflow is a meta-modeling approach for integrating dynamic parameters and run-time adaptation of parameters in a structured way into a certain class of dataflow models of computations, in particular, those models that have a well-defined concept of a graph *iteration* [6]. For example, SDF and cycle-static SDF (CSDF), which are discussed in [31], and multidimensional SDF (MDSDF), which is discussed in [37], have well defined concepts of iterations based on solutions to the associated forms of balance equations. Each of these models can be integrated with parameterized dataflow to provide a dynamically parameterizable form of the original model.

When parameterized dataflow is applied in this way to generalize a specialized dataflow model such as SDF, CSDF, or MDSDF, the specialized model is referred to as the *base model*, and the resulting, dynamically parameterizable form of the base model is referred to as *parameterized XYZ*, where *XYZ* is the name of the base model. For example, when parameterized dataflow is applied to SDF as the base model, the resulting model of computation, called *parameterized synchronous dataflow* (*PSDF*), is significantly more flexible than SDF as it allows arbitrary parameters of SDF graphs to be modified at run-time. Furthermore, PSDF provides a useful framework for quasi-static scheduling, where fixed-iteration looped schedules, such as single appearance schedules [7], for SDF graphs can be replaced by *parameterized looped schedules* [6, 40] in which loop iteration counts are represented as symbolic expressions in terms of variables whose values can be adapted dynamically through computations that are derived from the enclosing PSDF specification.

Intuitively, parameterized dataflow allows arbitrary attributes of a dataflow graph to be parameterized, with each parameter characterized by an associated domain of admissible values that the parameter can take on at any given time. Graph attributes that can be parameterized include scalar or vector attributes of individual actors, such as the coefficients of a finite impulse response filter or the block size associated with an FFT; edge attributes, such as the delay of an edge or the data type associated with tokens that are transferred across the edge; and graph attributes, such as those related to numeric precision, which may be passed down to selected subsets of actors and edges within the given graph.

The parameterized dataflow representation of a computation involves three cooperating dataflow graphs, which are referred to as the *body* graph, the *subinit* graph, and the *init* graph. The body graph typically represents the functional "core" of the overall computation, while the subinit and init graphs are dedicated to managing the parameters of the body graph. In particular, each output port of the subinit graph is associated with a body graph parameter such that data values produced at the output port are propagated as new parameter values of the associated parameter. Similarly, output ports of the init graph are associated with parameter values in the subinit and body graphs.

Changes to body graph parameters, which occur based on new parameter values computed by the init and subinit graphs, cannot occur at arbitrary points in time. Instead, once the body graph begins execution it continues uninterrupted through a graph iteration, where the specific notion of an iteration in this context can be specified by the user in an application-specific way. For example, in PSDF, the most natural, general definition for a body graph iteration would be a single *SDF iteration* of the body graph, as defined by the SDF repetitions vector [31].

However, an iteration of the body graph can also be defined as some constant number of iterations, for example, the number of iterations required to process a fixed-size block of input data samples. Furthermore, parameters that define the body graph iteration can be used to parameterize the body graph or the enclosing PSDF specification at higher levels of the model hierarchy, and in this way, the processing that is defined by a graph iteration can itself be dynamically adapted as the application executes. For example, the duration (or block length) for fixed-parameter processing may be based on the size of a related sequence of contiguous network

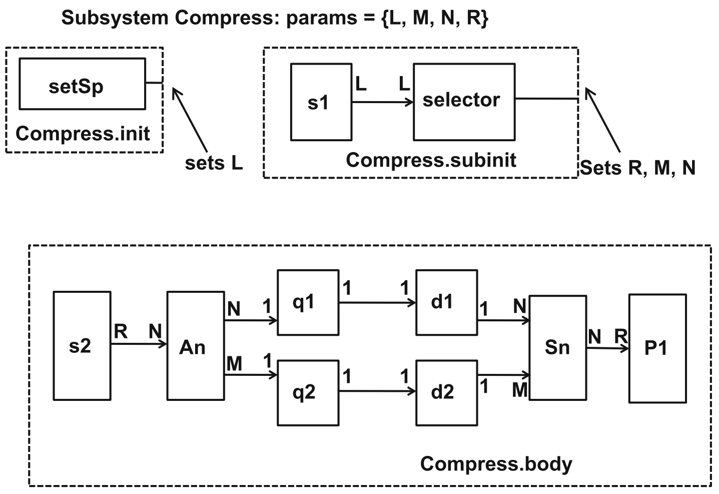


Fig. 2 An illustration of a speech compression system that is modeled using PSDF semantics. This illustration is adapted from [6]

packets, where the sequence size determines the extent of the associated graph iteration.

Body graph iterations can even be defined to correspond to individual actor invocations. This can be achieved by defining an individual actor as the body graph of a parameterized dataflow specification, or by simply defining the notion of iteration for an arbitrary body graph to correspond to the *next actor firing* in the graph execution. Thus, when modeling applications with parameterized dataflow, designers have significant flexibility to control the windows of execution that define the boundaries at which graph parameters can be changed.

A combination of cooperating body, init, and subinit graphs is referred to as a *PSDF specification*. PSDF specifications can be abstracted as PSDF actors in higher level PSDF graphs, and in this way, PSDF specifications can be integrated hierarchically.

Figure 2 illustrates a PSDF specification for a speech compression system. This illustration is adapted from [6]. Here *setSp* ("set speech") is an actor that reads a header packet from a stream of speech data, and configures L, which is a parameter that represents the length of the next speech instance to process. The *s1* and *s2* actors are input interfaces that inject successive samples of the current speech instance into the dataflow graph. The actor *s2* zero-pads each speech instance to a length R ($R \geq L$) so that the resulting length is divisible by N, which is the speech segment size. The *An* ("analyze") actor performs linear prediction on speech segments, and produces corresponding auto-regressive (AR) coefficients (in blocks of M samples), and residual error signals (in blocks of N samples) on its output edges. The actors *q1* and *q2* represent quantizers, and complete the modeling of the transmitter component of the body graph.

Receiver side functionality is then modeled in the body graph starting with the actors *d1* and *d2*, which represent dequantizers. The actor *Sn* ("synthesize") then reconstructs speech instances using corresponding blocks of AR coefficients and error signals. The actor *P1* ("play") represents an output interface for playing or storing the resulting speech instances.

The model order (number of AR coefficients) M, speech segment size N, and zero-padded speech segment length R are determined on a per-segment basis by the *selector* actor in the subinit graph. Existing techniques, such as the Burg segment size selection algorithm and AIC order selection criterion [32] can be used for this purpose.

The model of Fig. 2 can be optimized to eliminate the zero padding overhead (modeled by the parameter R). This optimization can be performed by converting the design to a parameterized cyclo-static dataflow (PCSDF) representation. In PCSDF, the parameterized dataflow meta model is integrated with CSDF as the base model instead of SDF.

For further details on this speech compression application and its representations in PSDF and PCSDF, the semantics of parameterized dataflow and PSDF, and quasi-static scheduling techniques for PSDF, see [6].

Parameterized cyclo-static dataflow (PCSDF), the integration of parameterized dataflow meta-modeling with cyclo-static dataflow, is explored further in [57]. The exploration of different models of computation, including PSDF and PCSDF, for the modeling of software defined radio systems is explored in [5]. In [36], Kee et al. explore the application of PSDF techniques to field programmable gate array implementation of the physical layer for 3GPP-Long Term Evolution (LTE). The integration of concepts related to parameterized dataflow in language extensions for embedded streaming systems is explored in [41]. General techniques for analysis and verification of hierarchically reconfigurable dataflow graphs are explored in [46].

5 Enable-Invoke Dataflow

Enable-invoke dataflow (EIDF) is another DSP-oriented dynamic dataflow modeling technique [51]. The utility of EIDF has been demonstrated in the context of behavioral simulation, FPGA implementation, and prototyping of different scheduling strategies [49–51]. This latter capability—prototyping of scheduling strategies—is particularly important in analyzing and optimizing embedded software. The importance and complexity of carefully analyzing scheduling strategies are high even for the restricted SDF model, where scheduling decisions have a major impact on most key implementation metrics [9]. The incorporation of dynamic dataflow features makes the scheduling problem more critical since application behaviors are less predictable, and more difficult to understand through analytical methods.

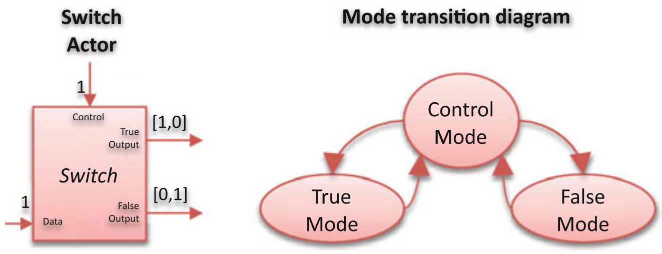
EIDF is based on a formalism in which actors execute through dynamic transitions among modes, where each mode has “synchronous” (constant production/consumption rate behavior), but different modes can have differing dataflow rates. Unlike other forms of mode-oriented dataflow specification, such as stream-based functions (see Sect. 8), SDF-integrated starcharts (see [15]), SysteMoc (see [15]), and CAL (see Sect. 3), EIDF imposes a strict separation between fireability checking (checking whether or not the next mode has sufficient data to execute), and mode execution (carrying out the execution of a given mode). This allows for lightweight fireability checking, since the checking is completely separate from mode execution. Furthermore, the approach improves the predictability of mode executions since there is no waiting for data (blocking reads)—the time required to access input data is not affected by scheduling decisions or global dataflow graph state.

For a given EIDF actor, the specification for each mode of the actor includes the number of tokens that is consumed on each input port throughout the mode, the number of tokens that is produced on each output port, and the computation (the *invoke function*) that is to be performed when the actor is invoked in the given mode. The specified computation must produce the designated number of tokens on each output port, and it must also produce a value for the *next mode* of the actor, which determines the number of input tokens required for and the computation to be performed during the next actor invocation. The next mode can in general depend on the current mode as well as the input data that is consumed as the mode executes.

At any given time between mode executions (actor invocations), an enclosing scheduler can query the actor using the *enable function* of the actor. The enable function can only examine the number of tokens on each input port (without consuming any data), and based on these “token populations”, the function returns a Boolean value indicating whether or not the next mode has enough data to execute to completion without waiting for data on any port.

The set of possible next modes for a given actor at a given point in time can in general be empty or contain one or multiple elements. If the next mode set is empty (the next mode is `null`), then the actor cannot be invoked again before it is somehow reset or re-initialized from environment that controls the enclosing dataflow graph. A `null` next mode is therefore equivalent to a transition to a mode that requires an infinite number of tokens on an input port. The provision for multi-element sets of next modes allows for natural representation of non-determinism in EIDF specifications.

When the set of next modes for a given actor mode is restricted to have at most one element, the resulting model of computation, called *core functional dataflow* (*CFDF*), is a deterministic, Turing complete model [51]. CFDF semantics underlie the *functional DIF* simulation environment for behavioral simulation of signal processing applications. Functional DIF integrates CFDF-based dataflow graph specification using the *dataflow interchange format* (*DIF*), a textual language for representing DSP-oriented dataflow graphs, and Java-based specification of intra-actor functionality, including specification of enable functions, invoke functions, and next mode computations [51].



Production & consumption behavior of modes

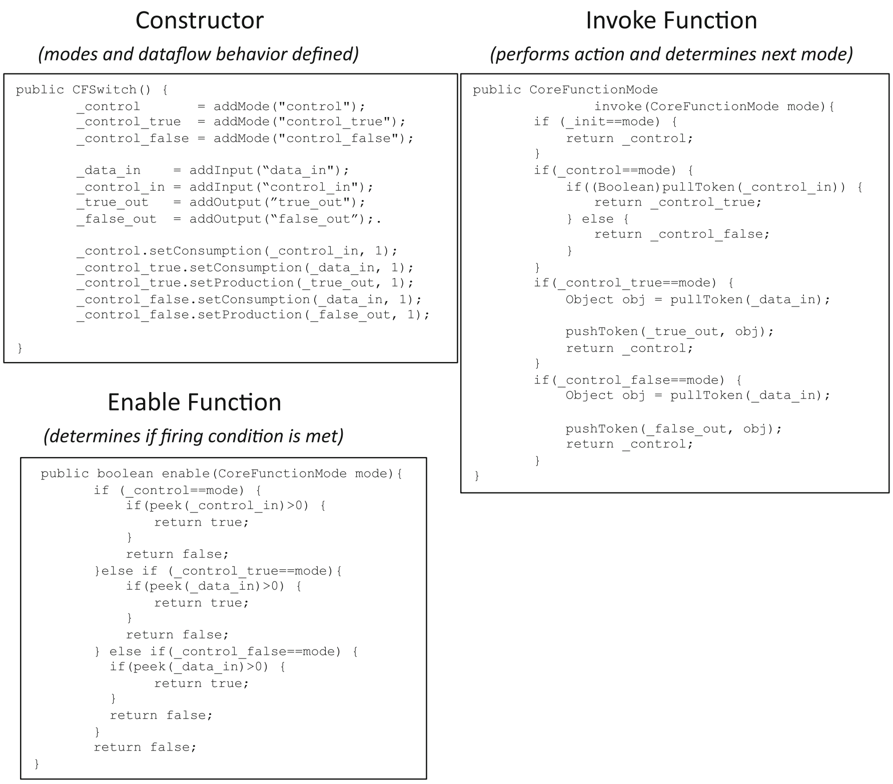
Mode	Control Input	Data Input	True Output	False Output
Control Mode	1	0	0	0
True Mode	0	1	1	0
False Mode	0	1	0	1

Fig. 3 An illustration of the design of a `switch` actor in CFDF

Figures 3 and 4 illustrate, respectively, the design of a CFDF actor and its implementation in functional DIF. This actor provides functionality that is equivalent to the Boolean dataflow switch actor described in Sect. 2.

6 Scenario Aware Dataflow

This section discusses Scenario-Aware Dataflow (SADF), which is a generalization of dataflow models with strict periodic behavior. Like most dataflow models, SADF is primarily a coordination language that highlights how actors (which are potentially executed in parallel) interact. To express dynamism, SADF distinguishes data and control explicitly, where the control-related coherency between the behavior (and hence, the resource requirements) of different parts of a signal processing application can be captured with so-called *scenarios* [26]. The scenarios commonly coincide with dissimilar (but within themselves more static) modes of operation originating, for example, from different parameter settings, sample rate conversion factors, or the signal processing operations to perform. Scenarios are typically defined by clustering operational situations with similar resource requirements [26]. The scenario-concept in SADF allows for more precise (quantitative) analysis results compared to applying traditional SDF-based analysis techniques. Still, common subclasses of SADF can be synthesized into efficient implementations [65].


Constructor
(modes and dataflow behavior defined)
Invoke Function
(performs action and determines next mode)
Enable Function
(determines if firing condition is met)

```
public CFSwitch() {
        _control       = addMode("control");
        _control_true  = addMode("control_true");
        _control_false = addMode("control_false");

        _data_in    = addInput(“data_in");
        _control_in = addInput(“control_in");
        _true_out   = addOutput(”true_out");
        _false_out  = addOutput(“false_out”);.

        _control.setConsumption(_control_in, 1);
        _control_true.setConsumption(_data_in, 1);
        _control_true.setProduction(_true_out, 1);
        _control_false.setConsumption(_data_in, 1);
        _control_false.setProduction(_false_out, 1);

}
```

```
public CoreFunctionMode
                invoke(CoreFunctionMode mode){
        if (_init==mode) {
            return _control;
        }
        if(_control==mode) {
            if((Boolean)pullToken(_control_in)) {
                return _control_true;
            } else {
                return _control_false;
            }
        }
        if(_control_true==mode) {
            Object obj = pullToken(_data_in);

            pushToken(_true_out, obj);
            return _control;
        }
        if(_control_false==mode) {
            Object obj = pullToken(_data_in);

            pushToken(_false_out, obj);
            return _control;
        }
}
```

```
public boolean enable(CoreFunctionMode mode){
       if (_control==mode) {
           if(peek(_control_in)>0) {
               return true;
           }
           return false;
       }else if (_control_true==mode){
           if(peek(_data_in)>0) {
               return true;
           }
           return false;
       } else if(_control_false==mode) {
         if(peek(_data_in)>0) {
               return true;
         }
         return false;
       }
       return false;
}
```

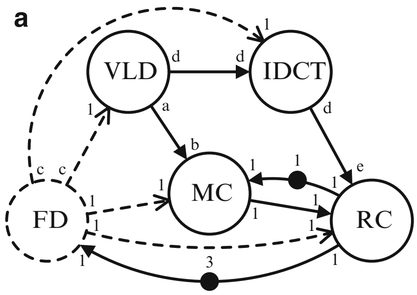
Fig. 4 An implementation of the `switch` actor design of Fig. 3 in the functional DIF environment

6.1 SADF Graphs

In this subsection SADF is introduced by some examples from the multi-media domain. Consider the MPEG-4 video decoder for the Simple Profile from [66, 70]. It supports video streams consisting of intra (I) and predicted (P) frames. For an image size of 176×144 pixels (QCIF), there are 99 macro blocks to decode for I frames and no motion vectors. For P frames, such motion vectors determine the new position of certain macro blocks relative to the previous frame. The number of motion vectors and macro blocks to process for P frames ranges between 0 and 99. The MPEG-4 decoder clearly shows variations in the functionality to perform and in the amount of data to communicate between the operations. This leads to large fluctuations in resource requirements [52]. The order in which the different situations occur strongly depends on the video content and is generally not periodic.

Figure 5 depicts an SADF graph for the MPEG-4 decoder in which nine different scenarios are identified. SADF distinguishes two types of actors: *kernels* (solid vertices) model the data processing parts, whereas *detectors* (dashed vertices)

a



b

Rate	(Sub)Scenario		
	I	P_0	P_x
a	0	0	1
b	0	0	x
c	99	1	x
d	1	0	1
e	99	0	x

$x \in \{30, 40, 50, 60, 70, 80, 99\}$

c

Actor	(Sub)Scenario	E(kCycles)
VLD	P_0	0
	All except P_0	40
IDCT	P_0	0
	All except P_0	17
MC	I, P_0	0
	P_{30}	90
	P_{40}	145
	P_{50}	190
	P_{60}	235
	P_{70}	265
	P_{80}	310
	P_{99}	390
RC	I	350
	P_0	0
	P_{30}, P_{40}, P_{50}	250
	P_{60}	300
	P_{70}, P_{80}, P_{99}	320
FD	All	0

Fig. 5 Modeling the MPEG-4 decoder with SADF. (**a**) Actors and channels; (**b**) parameterized rates; (**c**) worst-case execution times

control the behavior of actors through scenarios.[2] Moreover, *data* channels (solid edges) and *control* channels (dashed edges) are distinguished. Control channels communicate scenario-valued tokens that influence the control flow. Data tokens do not influence the control flow. The availability of tokens in channels is shown with a dot. Here, such dots are labeled with the number of tokens in the channel. The start and end points of channels are labeled with *production* and *consumption rates* respectively. They refer to the number of tokens atomically produced respectively consumed by the connected actor upon its *firing*. The rates can be fixed or scenario-dependent, similar as in PSDF. Fixed rates are positive integers. Parameterized rates are valued with non-negative integers that depend on the scenario. The parameterized rates for the MPEG-4 decoder are listed in Fig. 5b. A value of 0 expresses that data dependencies are absent or that certain operations are not performed in those scenarios. Studying Fig. 5b reveals that for any given scenario, the rate values yield a consistent SDF graph. In each of these scenario graphs, detector FD has a repetition vector entry of 1 [70], which means that scenario changes as prescribed by the behavior of detectors may occur only at iteration boundaries of each such scenario graph. This is not necessarily true for SADF in general as discussed below.

[2]In case of one detector, SADF literature may not show the detector and control channels explicitly.

SADF specifies execution times of actors (from a selected time domain, see Sect. 6.2) per scenario. Figure 5c lists the worst-case execution times of the MPEG-4 decoder for an ARM7TDMI processor. Figure 5b, c show that the worst-case communication requirements occur for scenario P_{99}, in which all actors are active and production/consumption rates are maximal. Scenario P_{99} also requires maximal execution times for VLD, IDCT, and MC, while for RC it is the scenario I in which the worst-case execution time occurs. Traditional SDF-based approaches need to combine these worst-case requirements into one (unrealistically) conservative model, which yields too pessimistic analysis results.

An important aspect of SADF is that sequences of scenarios are made explicit by associating *state machines* to detectors. The dynamics of the MPEG-4 decoder originate from control-flow code that (implicitly or explicitly) represents a state-machine with video stream content dependent guards on the transitions between states. One can think of if-statements that distinguish processing I frames from processing P frames. For the purpose of compile-time analysis, SADF abstracts from the content of data tokens (similar to SDF and CSDF) and therefore also from the concrete conditions in control-flow code. Different types of state machines can be used to model the occurrences of scenarios, depending on the compile-time analysis needs as presented in Sect. 6.2. The dynamics of the MPEG-4 decoder can be captured by a state-machine of nine states (one per scenario) associated to detector FD.

The operational behavior of actors in SADF follows two steps, similar to the *switch* and *select* actors in BDF. The first step covers the control part which establishes the mode of operation. The second step is like the traditional data flow behavior of SDF actors[3] in which data is consumed and produced. Kernels establish their scenario in the first step when a scenario-valued token is available on their control inputs. The operation mode of detectors is established based on external and internal forces. *Subscenario* denotes the result of the internal forces affecting the operation mode. External forces are the scenario-valued tokens available on control inputs (similar as for kernels). The combination of tokens on control inputs for a detector determine its scenario,[4] which (deterministically) selects a corresponding state-machine. A transition is made in the selected state machine, which establishes the subscenario. Where the scenario determines values for parameterized rates and execution time details for kernels, it is the subscenario that determines these aspects for detectors. Tokens produced by detectors onto control channels are scenario-valued to coherently affect the behavior of controlled actors, which is a key feature of SADF. Actor firings in SADF block until sufficient tokens are available. As a result, the execution of different scenarios can overlap in a pipelined fashion. For example, in the MPEG-4 decoder, IDCT is always ready to be executed immediately after VLD, which may already have accepted a control token with a

[3]Execution of the reflected function or program is enabled when sufficient tokens are available on all (data) inputs, and finalizes (after a certain execution time) with producing tokens on the outputs.

[4]If a detector has no control inputs, it operates in a default scenario ε and has one state machine.

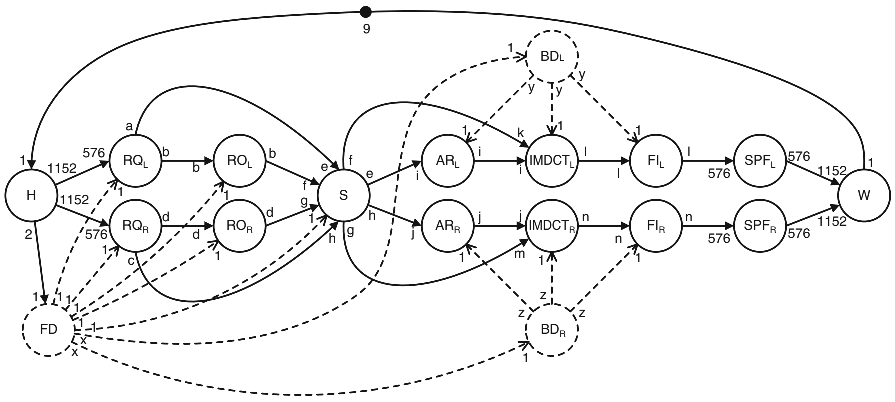


Fig. 6 Modeling an MP3 decoder with SADF using hierarchical control

different scenario value from FD. The ability to express such so-called *pipelined reconfiguration* is another key feature of SADF.

Now, consider the MP3 audio decoder example taken from [66] depicted in Fig. 6. It illustrates that SADF graphs can contain multiple detectors, which may even control each other's behavior. MP3 decoding transforms a compressed audio bitstream into pulse code modulated data. The stream is partitioned into frames of 1,152 mono or stereo frequency components, which are divided into two granules of 576 components structured in blocks [58]. MP3 distinguishes three frame types: Long (*L*), Short (*S*) and Mixed (*M*) and two block types: Long (*BL*) and Short (*BS*). A Long block contains 18 frequency components, while Short blocks include only 6 components. Long frames consist of 32 Long blocks, Short frames of 96 Short blocks and Mixed frames are composed of 2 Long blocks, succeeded by 90 Short blocks. The frame type and block type together determine the operation mode. Neglecting that the frame types and specific block type sequences are correlated leads to unrealistic models. The sequences of block types is dependent on the frame type, as is reflected in the structure of source code of the MP3 audio decoder. SADF supports *hierarchical control* to intuitively express this kind of correlation between different aspects that determine the scenario.

Figure 7a lists the parameterized rates for the MP3 decoder. Only five combinations of frame types occur for the two audio channels combined. A two-letter abbreviation is used to indicate the combined fame type for the left and right audio channel, respectively: *LL*, *SS*, *LS* and *SL*. Mixed frame type *M* covers both audio channels simultaneously. Detector FD determines the frame type with a state machine of five states, each uniquely identify a subscenario in $\{LL, SS, LS, SL, M\}$. The operation mode of kernel S depends on the frame types for both audio channels together and therefore it operates according to a scenario from this same set. The scenario of kernels $\mathrm{RQ_L}$, $\mathrm{RO_L}$ and $\mathrm{RQ_R}$, $\mathrm{RO_R}$ is only determined by the frame type for either the left or right audio channel. They operate in scenario *S*, *M* or *L* by receiving control tokens from FD, valued with either the left or right letter in *LL*, *SS*, *LS*, *SL* or with *M*.

a

Rate	Scenario		
	L	*S*	*M*
a, c	576	0	36
b, d	0	576	540

Rate	(Sub)Scenario				
	LL	*SS*	*LS*	*SL*	*M*
e	0	576	0	576	36
f	576	0	576	0	540
g	0	576	576	0	540
h	576	0	0	576	36
x	1	1	1	1	2

Rate	Scenario	
	BL	*BS*
i, j	18	0
k, m	0	6
l, n	18	6

Rate	SubScenario			
	LBL	*SBS*	*MBL*	*MBS*
y, z	32	96	2	90

b

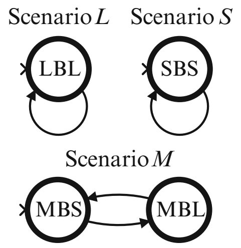


Fig. 7 Properties of the MP3 decoder model. (**a**) Parameterized rates; (**b**) state machines for BD_L and BD_R

Detectors BD_L and BD_R identify the appropriate number and order of Short and Long blocks based on the frame scenario, which they receive from FD as control tokens valued *L*, *S* or *M*. From the perspective of BD_L and BD_R, block types *BL* and *BS* are refinements (subscenarios) of the scenarios *L*, *S* and *M*. Figure 7b shows the three state machines associated with BD_L as well as BD_R. Each of their states implies one of the possible subscenarios in $\{\text{LBL}, \text{SBS}, \text{MBL}, \text{MBS}\}$. The value of the control tokens produced by BD_L and BD_R to kernels AR_L, $IMDCT_L$, FI_L and AR_R, $IMDCT_R$, FI_R in each of the four possible subscenarios matches the last two letters of the subscenario name (i.e., BL or BS). Although subscenarios LBL and MBL both send control tokens valued BL, the difference between them is the number of such tokens (similarly for subscenarios SBS and MBS).

Consider decoding of a Mixed frame. It implies the production of two *M*-valued tokens on the control port of detector BD_L. By interpreting each of these tokens, the state machine for scenario *M* in Fig. 7b makes one transition. Hence, BD_L uses subscenario MBL for its first firing and subscenario MBS for its second firing. In subscenario MBL, BD_L sends 2 *BL*-valued tokens to kernels AR_L, $IMDCT_L$ and SPF_L, while 90 *BS*-valued tokens are produced in subscenario MBS. As a result, AR_L, $IMDCT_L$ and SPF_L first process 2 Long blocks and subsequently 90 Short blocks as required for Mixed frames.

The example of Mixed frames highlights a unique feature of SADF: reconfigurations may occur *during* an iteration. An iteration of the MP3 decoder corresponds to processing frames, while block type dependent variations occur during processing Mixed frames. Supporting reconfiguration within iterations is fundamentally different from assumptions underlying other dynamic dataflow models, including for example PSDF. The concept is orthogonal to hierarchical control. Hierarchical control is also different from other dataflow models with hierarchy such as heterochronous dataflow [27]. SADF allows *pipelined* execution of the controlling and controlled behavior together, while other approaches commonly prescribe that the controlled behavior must first finish completely before the controlling behavior may continue.

6.2 Analysis

Various analysis techniques exist for SADF, allowing the evaluation of both qualitative properties (such as consistency and absence of deadlock) and best/worst-case and average-case quantitative properties (like minimal and average throughput).

Consistency of SADF graphs is briefly discussed now. The MPEG-4 decoder is an example of a class of SADF graphs where each scenario is like a consistent SDF graph and scenario changes occur at iteration boundaries of these scenario graphs (but still pipelined). Such SADF graphs are said to be *strongly consistent* [70], which is easy to check as it results from structural properties only. The SADF graph of the MP3 decoder does not satisfy these structural properties (for Mixed frames), but it can still be implemented in bounded memory. The required consistency property is called *weak consistency* [66]. Checking weak consistency requires taking the possible (sub)scenario sequences as captured by the state machines associated to detectors into account, which complicates a consistency check considerably.

Analysis of quantitative properties and the efficiency of the underlying techniques depend on the selected type of state machine associated to detectors as well as the chosen time model. For example, one possibility is to use non-deterministic state machines, which merely specify what sequences of (sub)scenarios *can* occur but not how often. This is typically used for best/worst-case analysis. Applying the techniques in [20, 23, 24] then allows computing that a throughput of processing 0.253 frames per kCycle can be guaranteed for the MPEG-4 decoder. An alternative is to use probabilistic state machines (i.e., Markov chains), which additionally capture the occurrence probabilities of the (sub)scenario sequences to allow for average-case analysis as well. Assuming that scenarios I, P_0, P_{30}, P_{40}, P_{50}, P_{60}, P_{70}, P_{80} and P_{99} of the MPEG-4 decoder may occur in any order and with probabilities 0.12, 0.02, 0.05, 0.25, 0.25, 0.09, 0.09, 0.09 and 0.04 respectively, the techniques in [67] allow computing that the MPEG-4 decoder processes on average 0.426 frames per kCycle. The techniques presented in [71] combine the association of Markov chains to detectors with exponentially distributed execution times to analyze the response time distribution of the MPEG-4 decoder for completing the first frame.

The semantics of SADF graphs where Markov chains are associated to detectors while assuming generic discrete execution time distributions[5] has been defined in [66] using Timed Probabilistic Systems (TPS). Such transition systems operationalize the behavior with states and guarded transitions that capture events like the begin and end of each of the two steps in firing actors and progress of time. In case an SADF graph yields a TPS with finite state space, it is amenable to analysis techniques for (Priced) Timed Automata or Markov Decision Processes and Markov Chains by defining reward structures as also used in (probabilistic) model checking. Theelen et al. [67] discusses that specific properties of dataflow models in

[5]This covers the case of constant execution times as so-called point distributions [66, 67].

general and SADF in particular allow for substantial state-space reductions during such analysis. The underlying techniques have been implemented in the SDF^3 tool kit [62], covering the computation of worst/best-case and average-case properties for SADF including throughput and various forms of latency and buffer occupancy metrics [68]. In case such exact computation is hampered by state-space explosion, [68, 70] exploit an automated translation into process algebraic models expressed in the Parallel Object-Oriented Specification Language (POOSL) [69], which allows for statistical model checking (simulation-based estimation) of the properties. The combination of Markov chains and exponentially distributed execution times has been studied in [71], using a process algebraic semantics based on Interactive Markov Chains [33] to apply a general-purpose model checker for analyzing response time distributions.

In case we abstract from the stochastic aspects of execution times and scenario occurrences, SADF is still amenable to worst/best-case analysis. Since SADF graphs are timed dataflow graphs, they exhibit *linear timing behavior* [20, 43, 76], which facilitates network-level worst/best-case analysis by considering the worst/best-case execution times for individual actors. For linear timed systems this is know to lead to the overall worst/best-case performance. For the class of strongly-consistent SADF graphs with a single detector (also called *FSM-based SADF*), very efficient performance analysis can be done based on a $(\max, +)$-algebraic interpretation of the operational semantics. It allows for worst-case throughput analysis, some latency analysis and can find critical scenario sequences without exploring the state-space of the underlying TPS. Instead, the analysis is performed by means of state-space analysis and maximum-cycle ratio analysis of the equivalent $(\max, +)$-automaton [20, 23, 24]. Geilen et al. [23] shows how this analysis can be extended for the case that scenario behaviors are not complete iterations of the scenario SDF graphs.

6.3 Synthesis

FSM-based SADF graphs have been extensively studied for implementation on (heterogeneous) multi-processor platforms [64]. Variations in resource requirements need to be exploited to limit resource usage without violating any timing requirements. The result of the design flow for FSM-based SADF implemented in the SDF^3 tool kit [62] is a set of Pareto optimal mappings that provide a trade-off in valid resource usages. For certain mappings, the application may use many computational resources and few storage resources, whereas an opposite situation may exist for other mappings. At run-time, the most suitable mapping is selected based on the available resources not used by concurrently running applications [59].

There are two key aspects of the design flow of [62, 64]. The first concerns mapping channels onto (possibly shared) storage resources. Like other dataflow models, SADF associates unbounded buffers with channels, but a complete graph may still be implemented in bounded memory. FSM-based SADF allows for

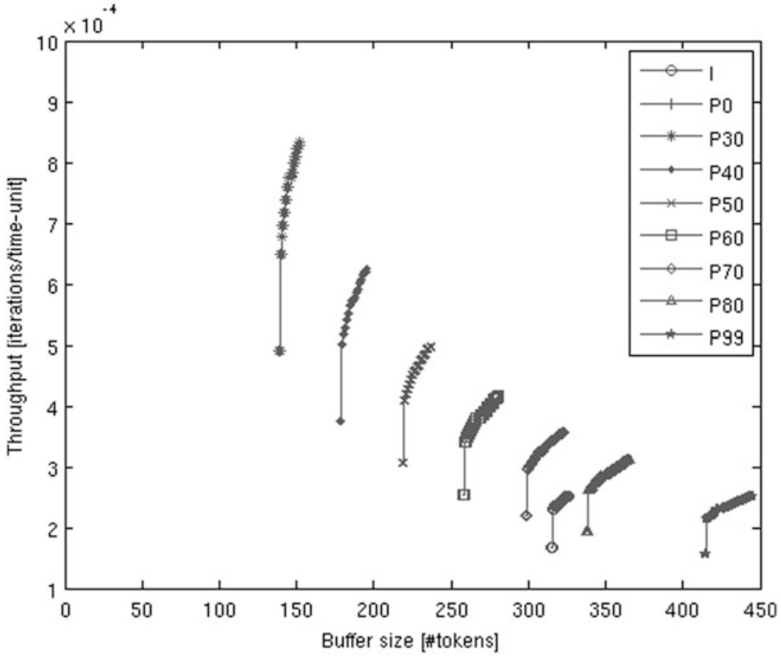


Fig. 8 Throughput/buffer size trade-off space for the MPEG-4 decoder

efficient compile-time analysis of the impact that certain buffer sizes have on the timing of the application. Hence, a synthesized implementation does not require run-time buffer management, thereby making it easier to guarantee timing. The design flow in [64] dimensions the buffer sizes of all individual channels in the graph sufficiently large to ensure that timing (i.e., throughput) constraints are met but also as small as possible to save memory and energy. It exploits the techniques of [63] to analyze the trade-off between buffer sizes and throughput for each individual scenario in the FSM-based SADF graph. After computing the trade-off space for all individual scenarios, a unified trade-off space for all scenarios is created. The same buffer size is assigned to a channel in all scenarios. Combining the individual spaces is done using Pareto algebra [22] by taking the free product of all trade-off spaces and selecting only the Pareto optimal points in the resulting space. Figure 8 shows the trade-off space for the individual scenarios in the MPEG-4 decoder. In this application, the set of Pareto points that describe the trade-off between throughput and buffer size in scenario P_{99} dominate the trade-off points of all other scenarios. Unifying the trade-off spaces of the individual scenarios therefore results in the trade-off space corresponding to scenario P_{99}. After computing the unified throughput/buffer trade-off space, the synthesis process in [64] selects a Pareto point with the smallest buffer size assignment that satisfies the throughput constraint as a means to allocate the required memory resources in the multiprocessor platform.

A second key aspect of the synthesis process is the fact that actors of the same or different applications may share resources. The set of concurrently active applications is typically unknown at compile-time. It is therefore not possible to construct a single static-order schedule for actors of different applications. The design flow in [64] uses static-order schedules for actors of the same application, but sharing of resources between different applications is handled by run-time schedulers with TDMA policies. It uses a binary search algorithm to compute the minimal TDMA time slices ensuring that the throughput constraint of an application is met. By minimizing the TDMA time slices, resources are saved for other applications. Identification of the minimal TDMA time slices works as follows. In [3], it is shown that the timing impact of a TDMA scheduler can be modeled into the execution time of actors. This approach is used to model the TDMA time slice allocation it computes. Throughput analysis is then performed on the modified FSM-based SADF graph. When the throughput constraint is met, the TDMA time slice allocation can be decreased. Otherwise it needs to be increased. This process continues until the minimal TDMA time slice allocation satisfying the throughput constraint is found.

7 Dynamic Polyhedral Process Networks

The chapter on *polyhedral process networks* (PPN) [73] deals with the automatic derivation of certain dataflow networks from *static affine nested loop programs* (SANLP). An SANLP is a nested loop program in which loop bounds, conditions and variable index expressions are (quasi-)affine expressions in the iterators of enclosing loops and static parameters.[6] Because many signal processing applications are not static, there is a need to consider *dynamic affine nested loop programs* (DANLP) which differ from SANLPs in that they can contain

1. *If-the-else* constructs with no restrictions on the condition [60].
2. *Loops* with no condition on the bounds [44].
3. *While* statements other than `while(1)` [45].
4. Dynamic parameters [78].

Remark. In all DANLP programs presented in subsequent subsections, arrays are indexed by affine functions of static parameters and enclosing for-loop iterators. This is why the *A* is still in the name.

[6]The corresponding tool is called PNgen [74], and is part of the `Daedalus` design framework [48], http://daedalus.liacs.nl.

```
%parameter N 8 16;

for i = 1:1:N,
  [x(i), t(i)] = F1(...);
end

for i = 1:1:N,
 if t(i) <= 0,
  [x(i)] = F2( x(i) );
 end
 [...] = F3( x(i) );
end
```

Fig. 9 Pseudo code of a simple weakly dynamic program

```
%parameter N 8 16;

for i = 1:1:N,
   ctrl(i) = N+1;
end
for i = 1:1:N,
 [out_0, out_1] = F1(...);
 [x_1(i)] = opd(out_0);
 [t_1(i)] = opd(out_1);
end

for i = 1:1:N,
 [t_1(i)] = ipd(t_1(i));
 if t_1(i) <= 0,
  [in_0] = ipd(x_1(i));
  [out_0] = F2(in_0);
  [x_2(i)] = opd(out_0);
  [ ctrl(i) ] = opd( i );
 end

 C = ipd( ctrl(i) );
 if i = C,
  [in_0] = ipd(x_2(C));
 else
  [in_0] = ipd(x_1(i));
 end

 [out_0] = F3(in_0);
 [...] = opd(out_0);
end
```

Fig. 10 Example of dynamic single assignment code

7.1 Weakly Dynamic Programs

While in a SANLP condition statements must be affine in static parameters and iterators of enclosing loops, if conditions can be anything in a DANLP. Such programs have been called *weakly dynamic programs* (WDP) in [60]. A simple example of a WDP is shown in Fig. 12.

The question of course is whether the argument of function $F3$ originates from the output of function $F2$ or function $F1$.

In the case of a SANLP, the input–output equivalent PPN is obtained by (1) converting the SANLP—by means of an *array analysis* [16, 17]—to a *single assignment code* (SAC) used in the compiler community and the systolic array community (see [34]); (2) deriving from the SAC a *polyhedral reduced dependence graph* [55] (PRDG); and (3) constructing the PPN from the PRDG [13, 39, 55].

While in a SAC every variable is written *only once*, in a *dynamic single assignment code* (dSAC) every variable is written *at most once*. For some variables, it is not known whether or not they will be read or written at compile time. For a WDP, however, not all dependences are known at compile time and, therefore, the analysis must be based on the so-called *fuzzy array dataflow analysis* [18]. This approach allows the conversion of a WDP to a dSAC. The procedure to generate the dSAC is out of the scope of this chapter. The dSAC for the WDP in Fig. 9 is shown in Fig. 10.

`C` in the dSAC shown in Fig. 10 is a parameter emerging from the *if*-statement in line 8 of the original program shown in Fig. 9. This *if*-statement also appears in the dSAC in line 14. The dynamic change of the value of `C` is accomplished by the lines 18 and 21 in Fig. 10. The control variable `ctrl(i)` in line 18 stores the iterations for which the data dependent condition that introduces `C` is true. Also, the variable `ctrl(i)` is used in line 21 to assign the correct value to `C` for the current iteration. See [60] for more details.

The dSAC can now be converted to two graph structures, namely the *Approximate reduced dependence graph* (ADG), and the *Schedule tree* (STree). The ADG is the dynamic counterpart of the static PRDG. Both the PRDG and the ADG are composed of processes N, input ports IP, output ports OP, and edges E [13, 55]. They contain all information related to the data dependencies between functions in the SAC and the dSAC, respectively. However, in a WDP some dependencies are not known at compile time, hence the name *approximate*. Because of this, the ADG has the additional notion of *linearly bounded set*, as follows.

Let be given four sets of functions $S1 = \{f_x^1(i) \mid x = 1..|S1|,\ i \in Z^n\}$, $S2 = \{f_x^2(i) \mid x = 1..|S2|,\ i \in Z^n\}$, $S3 = \{f_x^3(i) \mid x = 1..|S3|,\ i \in Z^n\}$, $S4 = \{f_x^4(i) \mid x = 1..|S4|,\ i \in Z^n\}$, an integral $m \times n$ matrix A and an integral n-vector b. A *linearly bounded set* (LBS) is a set of points $LBS = \{\ i \in Z^n \mid A.i \geq b,$

$$\begin{aligned}
&if\ S1 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S1|},\ f_x^1(i) \geq 0,\\
&if\ S2 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S2|},\ f_x^2(i) \leq 0,\\
&if\ S3 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S3|},\ f_x^3(i) > 0,\\
&if\ S4 \not\equiv \emptyset \Rightarrow \forall_{x=1..|S4|},\ f_x^4(i) < 0\ \}.
\end{aligned}$$

The set of points $B = \{\ i \in Z^n \mid A.i \geq b\ \}$ is called *linear bound* of the LBS and the set $S = S1 \cup S2 \cup S3 \cup S4$ is called *filtering set*. Every $f_x^j(i) \in S$ can be an arbitrary function of i.

Consider the dSAC shown in Fig. 10. The exact iterations i are not known at compile time because of the dynamic condition at line 14 in the dSAC (Fig. 10). That is why the notion of linearly bounded set is introduced, by which the unknown iterations i are approximated. So, ND_{N2} is the following LBS: $ND_{N2} = \{i \in Z \mid 1 \leq i \leq N \wedge 8 \leq N \leq 16,\ t\_1(i) \leq 0\}$. The linear bound of this LBS is the polytope $B = \{1 \leq i \leq N \wedge 8 \leq N \leq 16\}$ that captures the information known at compile time about the bounds of the iterations i. The variable $t\_1(i)$ is interpreted as an unknown function of i called filtering function whose output is determined at run time.

The STree contains all information about the execution order amongst the functions in the dSAC. The STree represents one valid schedule between all these functions called *global schedule*. From the STree a local schedule between any arbitrary set of the functions in the dSAC can be obtained by pruning operations on the STree. Such a local schedule may for example be needed when two or more processes are merged [61]. The STres is obtained by converting the dSAC to a syntax tree using a standard syntax parser, after which all the nodes and edges that are not related to nodes Fi (nodes $F1$, $F2$, and $F3$ in Fig. 10). See [60] for further details. A summary is depicted in Fig. 11.

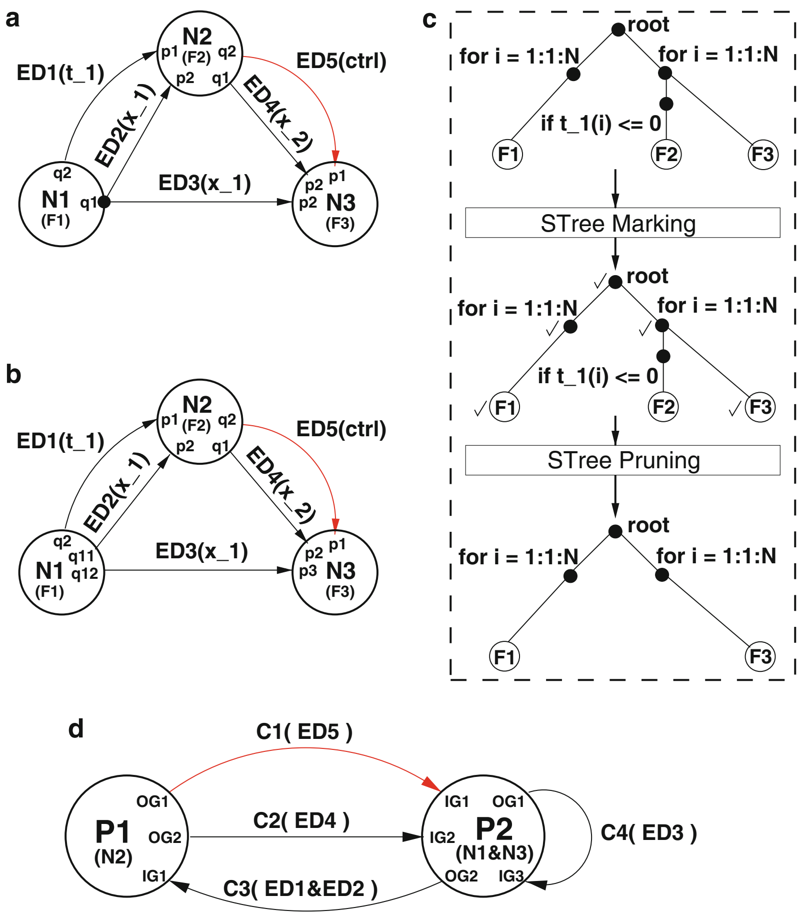


Fig. 11 Examples of (**a**) approximated dependence graph (ADG) model; (**b**) transformed ADG; (**c**) schedule tree and transformations; (**d**) process network model

The difference between the ADG in Fig. 11a and the transformed ADG in Fig. 11b is that an ADG may have several input ports connected to a single output port whilst in the transformed ADG every input port is connected to only one single output port (in accordance with the Kahn Process Network semantics [35]).

Parsing the STree in Fig. 11c top-down from left to right generates a program that gives a valid execution order (global schedule) among the functions $F1$, $F2$ and $F3$ which is the original order given by the dSAC.

The process network in Fig. 11d may be the result of a design space exploration, and some optimizations. For example, process $P2$ is constructed by grouping nodes $N1$ and $N3$ in the ADG in Fig. 11b. Because the behavior of process $P2$ is sequential (by default), it has to execute the functionality of nodes $N1$ and $N3$ in sequential order. This order is obtained from the STree in Fig. 11c. See [60] for details.

```
%parameter N 1 10;

 for j = 1 to N,
  for i = 1 to f(...),
   y[ i ] = F1()
  end
 end
 [...] = F2( y[5] ),
```

An example of a `Dynloop` program.

```
%parameter N 1 10;

for j = 1 to N,
  X[j] = f(...)
  for i = 1 to max_f,
    if i <= X[j],
    y[i] = F1()
    end
  end
 end
[] = F2( y[5] )
```

An equivalent Weakly Dynamic Program.

Fig. 12 A `Dynloop` program and its equivalent WDP program

In a (static) PPN, there are two models of FIFO communication [72], namely *in-order communication* and *out-of-order communication*. In the first model, the order in which tokens are read from a FIFO channel is the same as the order in which they have been written to the channel. In the second model that order is different. In a PPN that is input–output equivalent to a WDP, there are two more FIFO communication models, namely *in-order with coloring* and *out-of-order with coloring*. This is necessary because the number of tokens that will be written to a channel and read from that channel is not known at compile time. See [60] for details.

Buffer sizes can be determined using the procedure given in [73] and in [74], except that a conservative strategy (over-estimation) is needed due to the fact that the rate and the exact amount of data tokens that will be transferred over a particular data channel is unknown at compile-time. This can be done by modifying the iteration domains of all input/output ports, such that all dynamic *if*-conditions defining any of these iteration domains evaluate always to `true`.

7.2 Dynamic Loop-Bounds

Whereas in a SANLP loop bounds have to be affine functions of iterators of enclosing loops and static parameters, loop bounds in a DANLP program can be dynamic. Such programs have been called `Dynloop` programs in [44]. A simple example of a `Dynloop` program is shown at the left side in Fig. 12

A `Dynloop` program can be cast in the form of a WDP. See Sect. 7.1. The WDP corresponding to the `Dynloop` program at the left in Fig. 12 is shown at the right in Fig. 12.

```
%parameter N 1 10;

for j = 1 to N,
  X[j] = f()
  for i = 1 to max_f,
    if i <= X[j],
      y_1[j,i] = F1()
      ctrl_c1[i] = j
      ctrl_c2[i] = i
    end
    ctrl_c1_1[j,i] = ctrl_c1[i]
    ctrl_c2_1[j,i] = ctrl_c2[i]
  end
end
if max_f >= 5,
  c1 = ctrl_c1_1[N, 5]
  c2 = ctrl_c2_1[N, 5]
else
  c1 = N + 1
  c2 = max_f + 1
end
if c1 <= N & c2 == 5,
  in_0 = y_1[c1,c2]
else
  in_0 = 0
end
[...] = F2( in_0 )
```

Fig. 13 Final dSAC

The maximum value of $f()$, denoted by max_f, see line 5 at the right in Fig. 12 is substituted for the upper bound of the loop at line 4 at the left in Fig. 12. The value of max_f can be determined by studying the range of function $f()$.[7]

As in Sect. 7.1, a dynamic single assignment code (dSAC) can now be obtained by means of a fuzzy array dataflow analysis (FADA) [18]. This analysis introduces parameters to deal with the dynamic structure in the WDP. The values of these parameters have to be changed dynamically. This is done by introducing for every such parameter a control variable that stores the correct value of the parameter for every iteration. However, the straightforward introduction of control values as done in Sect. 7.1 violates the dSAC condition that every control variable is written *at most once*. To obtain a valid dSAC, an additional dataflow analysis for the control variables is necessary, resulting in additional control variables. See [44] for details.

The final dSAC is shown in Fig. 13 where it has been assumed that the variable y(5) has been initialized to zero.

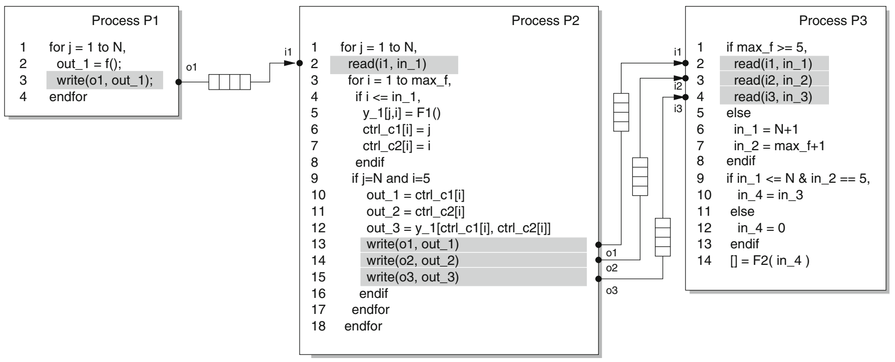
The control variables must be initialized with values that are greater than the maximum value of the corresponding parameters. For the example at hand, parameter $c1 \in [1..\mathtt{N}]$, and $c2 \in [1..\mathtt{max\_f}]$. Therefore, the corresponding control variables are initialized as follows:

$$\begin{aligned} \forall \mathtt{i} : 1 \le \mathtt{i} \le \mathtt{max\_f} : \mathtt{ctrl\_c1[i]} &= \mathtt{N} + 1, \\ \mathtt{ctrl\_c2[i]} &= \mathtt{max\_f} + 1. \end{aligned}$$

This initialization is not shown in Fig. 13 for the sake of brevity.

After applying the standard *linearization* [72], and its extension described in Sect. 7.1, and estimating buffer sizes as described in that same subsection, the resulting PPN is as shown in Fig. 14.

[7]If that is not possible, then an alternative way to estimate max_f is given in [44].



```
for j = 1 to N,
  out_1 = f();
  write(o1, out_1);
endfor
```

```
for j = 1 to N,
  read(i1, in_1)
  for i = 1 to max_f,
    if i <= in_1,
      y_1[j,i] = F1()
      ctrl_c1[i] = j
      ctrl_c2[i] = i
    endif
    if j=N and i=5
      out_1 = ctrl_c1[i]
      out_2 = ctrl_c2[i]
      out_3 = y_1[ctrl_c1[i], ctrl_c2[i]]
      write(o1, out_1)
      write(o2, out_2)
      write(o3, out_3)
    endif
  endfor
endfor
```

```
if max_f >= 5,
  read(i1, in_1)
  read(i2, in_2)
  read(i3, in_3)
else
  in_1 = N+1
  in_2 = max_f+1
endif
if in_1 <= N & in_2 == 5,
  in_4 = in_3
else
  in_4 = 0
endif
[] = F2( in_4 )
```

Fig. 14 The final PPN derived from the program in Fig. 13

7.3 *Dynamic While-Loops*

Whereas in a SANLP program only `while(1)` loops are allowed, in a DANLP program any while-loop is acceptable. Such DANLP programs have been called *while-loop affine programs* (WLAP) in [45].

There are a number of publications that address the problem of while loops parallelization [4, 10, 12, 25, 28, 29, 53, 54]. The approach presented here has the advantage that it

- Supports both task-level and data-level parallelism.
- Generates also parallel code for multi-processor systems having distributed memory.
- Provides an automatic data-dependence analysis procedure.
- Exposes and utilizes all available parallelism.

An example is shown at the left side in Fig. 15.

Again, the question is from where, say, function $F7$ gets its scalar argument `x`. Because this is not known at compile-time, a *fuzzy array dataflow analysis* (FADA) [18] is necessary to find all data dependencies.

The approach to convert a WLAP program to an input–output equivalent *polyhedral process network* (PPN) goes in four steps. First, all data-dependency relations in the initial WLAP program have to be found by applying the FADA analysis on it. Recall that the result of the analysis is approximated, i.e., it depends on parameters which values are determined at run-time. Second, based on the results of the analysis, the initial WLAP is transformed into a *dynamic Single Assignment Code* (dSAC) representation. See Sect. 7.1. Parameters that are introduced by the FADA appear in the dSAC, and their values are assigned using control variables. Third, the control variables are generated in a way that extends the methods in Sects. 7.1 and 7.2 to be applicable for WLAP programs as well, see [45]. Fourth, the topology of the corresponding PPN is derived as well as the code to be executed in the processes of the PPN.

```
1   &parameter EPS 0.005

2   for i = 1 to N,
3     y[i] = F1()
4     x = F2( y[i] )
5    while ( x >= EPS )
6      x = F3()
7      for j = i+1 to N+1,
8        y[j] = F4( y[j-1] )
9        x = F5( x, y[j] )
10     end
11      y[i] = F6( x )
12    end
13     out = F7( x )
14   end
```

An example of a WLAP program

```
1   %parameter EPS 0.005

2   w = 0
3   ctrl_x_5 = (N+1,0)
4   for i = 1 to N,
5     y_1[i] = F1()
6     in_2 = y_1[i]
7     x_2[i] = F2( in_2 )
8     while (in_w = $\sigma_x(\langle W‹(i‹w)\rangle)$ >= EPS),
9       w = w + 1
11      x_3[i,w] = F3()
11      for j = i+1 to N+1,
12        in_4 = $\sigma_y(\langle S_4‹(i‹w‹j)\rangle)$
13        y_4[i,w,j] = F4( in_4 )
14        in_5_x = $\sigma_x(\langle S_5‹(i‹w‹j)\rangle)$
15        in_5_y = y_4[i,w,j]
16        x_5[i,w,j] = F5( in_5_x, in_5_y )
17        ctrl_x_5 = (i,w)
18       end
19      in_6 = $\sigma_x(\langle S_6‹(i‹w)\rangle)$
20      y_6[i,w] = F6( in_6 )
21    end
22    ctrl_x_5_[i] = ctrl_x_5
23    ($\alpha$‹$\beta$) = ctrl_x_5_[i]
24    in_7 = $\sigma_x(\langle S_7‹(i‹\alpha‹\beta)\rangle)$
25    out = F7( in_7 )
26  end
```

The corresponding final dSAC

Fig. 15 An example of a while-loop affine program and its corresponding dynamic single assignment program

The iterator w is associated with the while loop and is initialized with value 0, meaning that the while loop has never been executed. The parameter α captures the value of the for-loop iterator in the enclosing while-loop and is initialized to $N+1$. The parameter β is the upper bound of the while-loop iterator w. Because $\alpha \in [1..\mathtt{N}]$ and $\beta \geq 1$, the above initializations satisfy the condition that their values are never taken by the corresponding parameters. From line `23` at the right side in Fig. 15, it follows that the control variable `ctrl_x_5` is initialized to `ctrl_x_5 = (N+1,0)` at line `3` at the right side in Fig. 15. Where does the control variable `ctrl_x_5` come from? It comes from the construction of the dSAC. The procedure to derive the final dSAC is largely based on [18] and its extension in Sect. 7.2. The problem is again that the dSAC resulting from the FADA analysis is not a proper dSAC because it violates the property that every variable is written *at most once*. The relation between writing to and reading from the control variables must be identified by performing a dataflow analysis for the control variables, where the writings to them occur inside a while-loop. To that end, an additional control variable `ctrl_x_5_` is introduced right *after* the while-loop, see line `22` at the right in Fig. 15. The new control variable is written at every iteration of *for*-loop `i` and takes the value either of control variable `ctrl_x_5` assigned on the last iteration of the while-loop, or its initial value, if the while-loop is not executed. A *static* exact

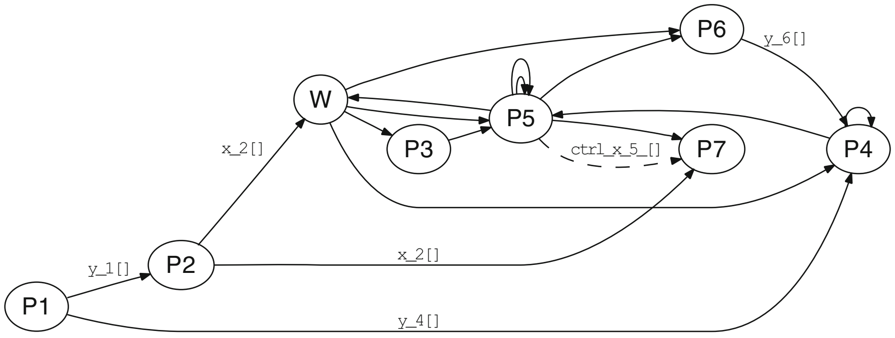


Fig. 16 The PPN for the program in Fig. 15

```
%parameter EPS 0.005

w = 0
for i = 1 to N,
  while(1),
    w = w + 1
    if (w > 2) then w = 2
    if (w == 1),
      read(P2, 1, in_w)
    else
      read(P5, 2, in_w)
    end
    out_w = (in_w >= EPS)
    write(P3, 3, out_w)
    write(P4, 4, out_w)
    write(P5, 5, out_w)
    write(P6, 6, out_w)
    if (!out_w) <break>
  end
end
```

Code of process W

```
w = 0
ctrl_x_5 = (N+1,0)
for i = 1 to N,
  while(1),
    w = w + 1
    if (w > 2) then w = 2
    read(W, 1, in_w)
    if (!in_w) <break>
    for j = i+1 to N+1,
      if (j == i+1),
        if (w == 1),
          read(P3, 2, in_5_x)
        else
          read(P5, 3, in_5_x)
        en
      else
        read(P5, 4, in_5_x)
      end
      read(P4,5, in_5_y)
      out_5 = F5( in_5_x, in_5_y )
      ctrl_x_5 = (i,w)
      if (j == N+1),
        write(P5, 6, out_5)
      else
        write(P5, 7, out_5)
      endif
    end
  end
  out_5_c = ctrl_x_5
  out_5_x = out_5
  write(P7, 8, out_5_c)
  write(P7, 9, out_5_x)
end
```

Code of process P5

```
w = 0
for i = 1 to N,
  read(P5, 1, in_c)
  if (in_c.β>=1 && 1<= in_c.α <= i),
     read(P5, 2, in_7)
  else
     read(P2, 3, in_7)
  end
  out = F7( in_7 )
end
```

Code of process P7

Fig. 17 Processes W, $P5$, and $P7$ after linearization

array dataflow analysis (EADA) [16] can be performed on this new control variable `ctrl_x_5_`. This is possible because the new control variable is not surrounded by the dynamic while-loop, i.e., it is outside the while loop.

The PPN that corresponds to the final dSAC in Fig. 15 is depicted in Fig. 16.

This PPN consists of 8 processes and 18 channels. The processes $P1$–$P7$ correspond to the functions $F1$–$F7$ in Fig. 15. Process W corresponds to the while condition at line 8 of the final dSAC in Fig. 15

The code for processes W, $P5$, and $P7$ is shown in Fig. 17. Process W is an example of a process detecting the termination of the while-loop at line 5 at the left in Fig. 15. Process $P5$ is an example of a process executing a function enclosed in the while-loop. Process $P7$ is an example of a process that runs a function *outside* the while-loop, and has a data dependency with a function *inside* the while-loop.

7.4 Parameterized Polyhedral Process Networks

Parameters that appear in a SANLP program are static. In a DANLP, parameters can be dynamic. A polyhedral process network [73] that is input–output equivalent to such a DANLP program is, then, a *parameterized polyhedral process network* called P^{3}N in [78].

Remark. There are two assumptions here. First, dynamic conditions, dynamic loop bounds and dynamic while-loops are left out to focus only on dynamic parameters. Second, values of the dynamic parameters are obtained from the environment.

The formal definition of a P^{3}N is given in [78], and is only slightly different from the definition given in [73]. Although the consistency of a P^{3}N has to be checked at run-time, still some analysis can be done at compile-time. A simple example of a P^{3}N is shown in Fig. 18.

Figure 18a is a static PPN, process *P3* of which is shown in Fig. 18b. Figure 18c is a P^{3}N version of the PPN in Fig. 18a. Process *P*3 of the P^{3}N in Fig. 18c is shown in Fig. 18d. The PPN and the P^{3}N have the same *dataflow* topology. Processes *P2* and *P3* in the P^{3}N in Fig. 18c are reconfigured by two parameters *M* and *N* whose values are updated from *the environment* at run-time using process *Ctrl* and FIFO channels *ch7*, *ch8*, and *ch9*. The P^{3}N shown in Fig. 18c may be derived from a sequential program, yet it can also be constructed from library elements as in [31].

Recall from [73] that a parametric polyhedron $\mathscr{P}(\mathbf{p})$ is defined as $\mathscr{P}(\mathbf{p}) = \{(w, x_1, \ldots, x_d) \in \mathbb{Q}^{d+1} \mid A \cdot (w, x_1, \ldots, x_d)^T \geq B \cdot \mathbf{p} + b\}$ with $A \in \mathbb{Z}^{m \times d}, B \in \mathbb{Z}^{m \times n}$ and $c \in \mathbb{Z}^m$. For nested loop programs, w is to be interpreted as the one-dimensional `while(1)` index, and d as the depth of a loop nest. For a particular value of w the polyhedron gets closed, i.e., it becomes a polytope. The parameter vector $\mathbf{p}$ is bounded by a polytope $\mathscr{P}_{\mathbf{p}} = \{\mathbf{p} \in \mathbb{Q}^n \mid C \cdot \mathbf{p} \geq d\}$.

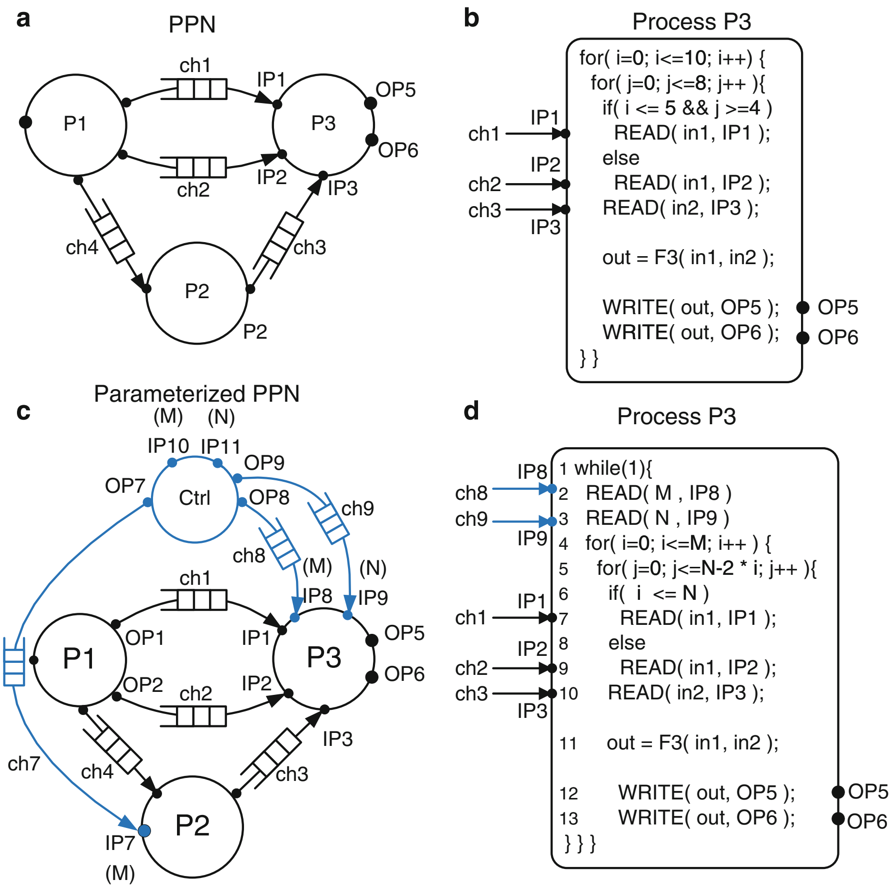
The domain D_P of a process is defined as the set of all integral points in its underlying parametric polyhedron, i.e., $D_P = \mathscr{P}_P(\mathbf{p}) \cap \mathbb{Z}^{d+1}$. The domains D_{IP} and D_{OP} of an input port *IP* and an output port *OP*, respectively, of a process are subdomains of the domain of that process.

The following four notions play a role in the operational semantics of a P^{3}N:

- Process iteration.
- Process cycle.
- Process execution.
- Quiescent point.

A **process iteration** of process P is a point $(w, x_1, \ldots, x_d) \in D_P$, where the following operations are performed sequentially: reading a token from each IP for which $(w, x_1, \ldots, x_d) \in D_{IP}$, executing process function F_P, and writing a token to each OP for which $(w, x_1, \ldots, x_d) \in D_{OP}$.

A **process cycle** $CYC_P(\mathscr{S}, \mathbf{p}) \subset D_P$ is the set of lexicographically ordered points $\in D_P$ for a particular value of $w = \mathscr{S} \in \mathbb{Z}^+$. The lexical ordering is typically imposed by a loop nest.



```
for( i=0; i<=10; i++) {
  for( j=0; j<=8; j++ ){
    if( i <= 5 && j >=4 )
      READ( in1, IP1 );
    else
      READ( in1, IP2 );
    READ( in2, IP3 );

    out = F3( in1, in2 );

    WRITE( out, OP5 );
    WRITE( out, OP6 );
} }
```

```
while(1){
  READ( M , IP8 )
  READ( N , IP9 )
  for( i=0; i<=M; i++ ) {
    for( j=0; j<=N-2 * i; j++ ){
      if( i <= N )
        READ( in1, IP1 );
      else
        READ( in1, IP2 );
      READ( in2, IP3 );

      out = F3( in1, in2 );

      WRITE( out, OP5 );
      WRITE( out, OP6 );
}}}
```

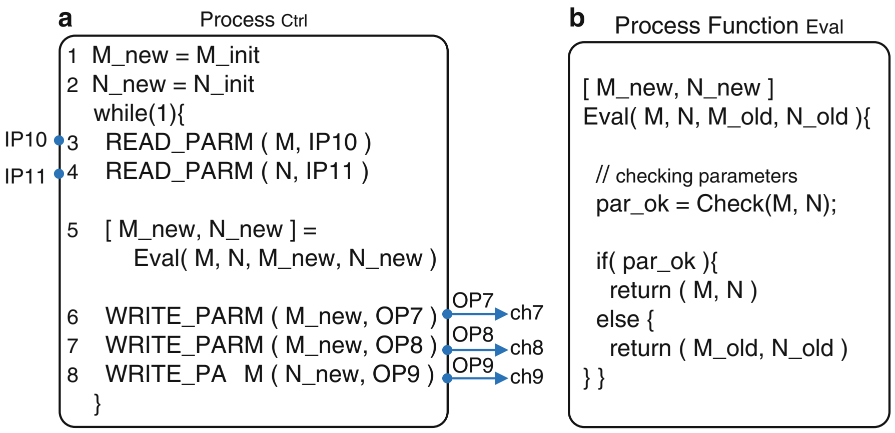
Fig. 18 (**a**) An example of a PPN, (**b**) process *P3* in the PPN, (**c**) an example of a P^{3}N, and (**d**) process *P3* in the P^{3}N

A **Process execution** E_P is a sequence of process cycles denoted by $CYC_P(1,\mathbf{p}_1) \rightarrow CYC_P(2,\mathbf{p}_2) \rightarrow \ldots \rightarrow CYC_P(k,\mathbf{p}_k)$, where $k \rightarrow \infty$.

A point $Q_P(\mathscr{S},\mathbf{p}_i) \in CYC_P(\mathscr{S},\mathbf{p}_i)$ of process P is a **quiescent point** if $CYC_P(\mathscr{S},\mathbf{p}_i) \in E_P$ and $\neg(\exists(w,x_1,\ldots,x_d) \in CYC_P(\mathscr{S},\mathbf{p}_i) : (w,x_1,\ldots,x_d) \prec Q_P(\mathscr{S},\mathbf{p}_{\mathscr{S}})$.

Thus, process P can change parameter values at the first process iteration of any process cycle during the execution. The notion of quiescent points as being the points at which values of the parameters **p** can change appears also in [47]. The behavior of the control process *Ctrl* is given in Fig. 19a.

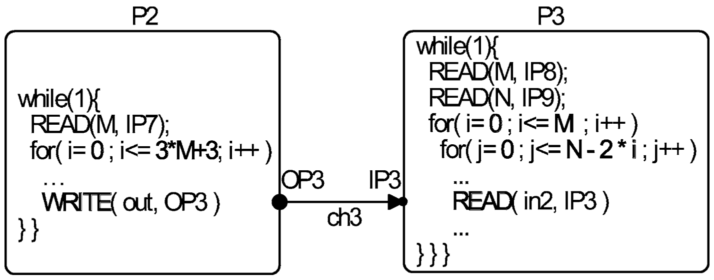
Process *Ctrl* starts with at least one valid parameter combination (lines `1--2`) and then reads parameters from the environment (lines `3--4`) every pre-specified time interval. For every incoming parameter combination, the process function `Eval` (line `5`) checks whether the combination of parameter values is valid. The



```
1 M_new = M_init
2 N_new = N_init
  while(1){
3   READ_PARM ( M, IP10 )
4   READ_PARM ( N, IP11 )

5   [ M_new, N_new ] =
       Eval( M, N, M_new, N_new )

6   WRITE_PARM ( M_new, OP7 )
7   WRITE_PARM ( M_new, OP8 )
8   WRITE_PA  M ( N_new, OP9 )
  }
```

```
[ M_new, N_new ]
Eval( M, N, M_old, N_old ){

  // checking parameters
  par_ok = Check(M, N);

  if( par_ok ){
    return ( M, N )
  else {
    return ( M_old, N_old )
} }
```

Fig. 19 (**a**) Control process *Ctrl* and (**b**) process function *Eval*



```
while(1){
  READ(M, IP7);
  for( i=0 ; i<= 3*M+3; i++ )
    ...
    WRITE( out, OP3 )
}}
```

```
while(1){
  READ(M, IP8);
  READ(N, IP9);
  for( i=0 ; i<= M ; i++ )
    for( j=0 ; j<= N - 2 * i ; j++ )
      ...
      READ( in2, IP3 )
      ...
}}}
```

Fig. 20 Which combinations (M, N) do ensure consistency of P^{3}N?

implementation of function `Eval` is given in Fig. 19b. If the combination is valid, then function `Eval` returns the current parameter values (`M, N`). Otherwise, the last valid parameters combination (propagated through `M_new, N_new` in this example) is returned. After the evaluation of the parameters combination, process *Ctrl* writes the parameter values to output ports (lines 6–8) when all channels *ch7*, *ch8*, and *ch9* have at least one buffer place available. When at least one channel buffer is full, the incoming parameters combination is discarded and the control process continues to read the next parameters combination from the environment. Furthermore, the depth of the FIFOs of the control channels determines how many process cycles of the dataflow processes are allowed to overlap.

Valid parameter values lead to the consistent execution of a P^{3}N, i.e., without deadlocks and with bounded memory (FIFOs with finite capacity). To illustrate the problem, consider channel *ch3* connecting processes *P2* and *P3* of the P^{3}N given in Fig. 18c.

The access of processes *P2* and *P3* to channel *ch3* is depicted in Fig. 20.

Consistency requires that, for each corresponding process cycle of both processes $CYC_{P2}(i,M_i)$ and $CYC_{P3}(i,M_i,N_i)$, the number of tokens produced by process *P2* to channel *ch3* must be equal to the number of tokens consumed by process *P3* from channel *ch3*. For example, if $(M,N) = (7,8)$, *P2* produces 25 tokens to *ch3* and *P3* consumes 25 tokens from the same channel after one corresponding process cycle of both processes. It can be verified that *P2* produces 13 tokens to *ch3* while *P3* requires 20 tokens from it in a corresponding process cycle when $(M,N) = (3,7)$. Thereby, in order to complete one execution cycle of *P3* in this case, it will read data from *ch3* which will be produced during the next execution cycle of *P2*. Evidently this leads to an incorrect execution of the P^{3}N. From this example, it is clearly seen that the incoming values of (M,N) must satisfy certain relation to ensure the consistent execution of the P^{3}N.

Although the consistency of a P^{3}N has to be checked at run-time, still some analysis can be done at design-time. This is because input ports and output ports of a process cycle are parametric polytopes. The number of points in a port domain equals the number of tokens that will be written to a channel or read from a channel depending on whether the port is an output port or an input port, respectively. The condition $|D_{OP}^{CYC}| = |D_{IP}^{CYC}|$ can be checked by comparing the number of points in both port domains. The counting problem can be solved in polynomial time using the *Barvinok* library [73, 75]. In general the number of points in domain $D_X = \mathscr{P}_X(\mathbf{p}) \cap \mathbb{Z}^{d+1}$, where X stand for either a process *P*, an input port *IP*, or an output port *OP*, is a set of quasi-polynomials [73].

For the example shown in Fig. 20, the difference $|D_{OP}^{CYC}| - |D_{IP}^{CYC}|$ is,

$$\begin{cases} (1+N+N\cdot M-M^2)-(3M+4)=0 & \text{if } (M,N)\in C1 \\ (1+\frac{3}{4}N+\frac{1}{4}N^2+\frac{1}{4}N-\frac{1}{4}\cdot\{0,1\}_N)-(3M+4)=0 & \text{if } (M,N)\in C2 \end{cases}$$

where $C1 = \{(M,N) \in \mathbb{Z}^2 \mid M \le N \wedge 2M \ge 1+N\}$, $C2 = \{(M,N) \in \mathbb{Z}^2 \mid 2M \le N\}$, and $\{0,1\}_N$ is a periodic coefficient with period 2.[8] In this example, if the range of the parameters is $0 \le M,N \le 100$, then there are only ten valid parameter combinations. If $0 \le M,N \le 1{,}000$, then there are 34 valid parameter combinations, and if $0 \le M,N \le 10{,}000$, then the number of valid combinations is 114.

The symbolic subtraction of the quasi-polynomials can result in constant zero, non-zero constant, or a quasi-polynomial. In the first case, consistency is always preserved for all parameters within the range. In the second case, all parameters within the range are invalid, because they violate the consistency condition. In the third case, a quasi-polynomial remains, and only some parameter combinations within the range are valid for the consistency condition. The equations can be solved at design time, and all valid parameter combinations are put in a table which is stored in a function *Check*. At run-time, the control process only propagates those incoming parameter combinations that match an entry in the table. Alternatively, function *Check* evaluates the difference between the two quasi-polynomials against

[8] $\{0,1\}_N$ is `0` or `1` depending on whether N is even or odd, respectively.

zero with incoming parameter values at run-time. When using a table, the execution time of the P^3N is almost equal to the execution time of the corresponding PPN. On the other hand, evaluation the polynomials at run-time overlaps the dataflow processing. For medium and high workloads (execution latency of the processes) the overhead is negligible. See [78] for further details.

8 The Stream-Based Function Model

The active entities in dataflow graphs are pure functions, embedded in *Actors*. In dataflow (polyhedral and parameterized polyhedral) process networks, the active entities encompass state and pure functions. They are called *Processes*. Actors and processes are specified in a *host language* like C, C++ or Java, by convention.

Remark. For reasons of convenience, the active entities in dataflow graphs and dataflow networks will in this section be called *actors*, whether they are void of state, encompass a single thread of control, or are processes. Thus static dataflow graphs [31], dynamic dataflow graphs, polyhedral process [73] and Sect. 7.4, and Kahn process networks [21] will collectively be referred to as *Dataflow Actor Networks* (DFAN). They obey the Kahn coordination semantics, possibly augmented with actor firing rules, annotated with deadlock free minimal FIFO buffer capacities, and one or more global schedules associated with them.

This section deals with a parallel model for actors, called *Stream-based function* (SBF) in [38] SBF is appealing when it comes to implementing a DFAN in a heterogeneous multiprocessor execution platform that consists of a number of computational elements (processors), and a communication, synchronization, and storage infrastructure. This requires a *mapping* that relates an untimed application model and a timed architecture model together: Actors are assigned to processors (both of type ISA and dedicated), FIFO channels are logically embedded in—often distributed—memory, and DFAN communication primitives and protocols are transformed to platform specific communication primitives and protocols. The specification of platform processors and DFAN actors may be quite different.

The SBF can serve as an intermediate specification between the conventional DFAN actor specification, and a variety of computational elements in the execution platform.

8.1 The Stream-Based Function Actor

The SBF is composed of a *set of functions*, called *function repertoire*, a *transition and selection function*, called *controller*, and a combined function and data *state*, called *private memory*. The controller selects a function from the function repertoire that is associated with a *current function state*, and makes a transition to the *next*

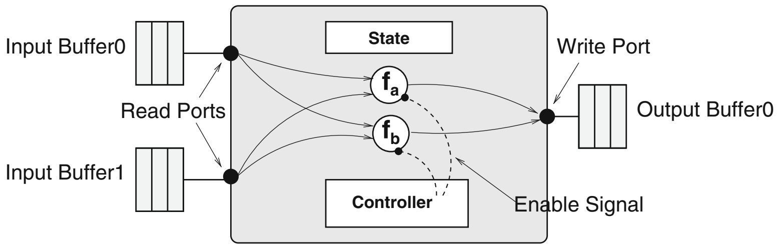


Fig. 21 A simple SBF actor

function state. A selected function is *enabled* when all its input arguments can be read, and all its results can be written; it is blocked otherwise. A selected non-blocked function must evaluate or *fire*. Arguments and results are called *input tokens* and *output tokens*, respectively.

An actor operates on sequences of tokens, streams or signals, as a result of a repetitive enabling and firing of functions from the function repertoire. Tokens are *read* from external ports and/or loaded from private memory, and are *written* to external ports and/or stored in private memory. The external ports connect to FIFO channels through which actors communicate point-to-point.

Figure 21 depicts an illustrative stream-based function (SBF) actor. Its function repertoire is $P = \{f_{init}, f_a, f_b\}$. The two read ports, and the single write port connect to two input and one output FIFO channels, respectively. P contains at least an initialization function f_{init}, and is a finite set of functions. The functions in P must be selected in a mutual exclusive order. That order depends on the controller's selection and transition function. The private memory consists of a function state part, and a data state part that do not intersect.

8.2 The Formal Model

If C denotes the SBF actor's function state space, and if D denotes its data state space, then the actor's state space S is the Cartesian product of C and D,

$$S = C \times D, \qquad C \cap D = \emptyset. \tag{2}$$

Let $c \in C$ be the current function state. From c, the controller selects a function and makes a function state transition by activating its selection function μ and transition function ω as follows,

$$\mu : C \to P, \qquad \mu(c) = f, \tag{3}$$

$$\omega : C \times D \to C, \qquad \omega(c,d) = c'. \tag{4}$$

The evaluation of the combined functions μ and ω is instantaneous. The controller cannot change the content of C; it can only observe it. The transition function allows for a dynamic behavior as it involves the data state space D. When the transition function is only a map from C to C, then the trajectory of selected functions will be static (see [31, 73]). Assuming that the DFAN does not deadlock, and that its input streams are not finite, the controller will repeatedly invoke functions from the function repertoire P in an endless firing of functions,

$$f_{init} \xrightarrow{\mu(\omega(S))} f_a \xrightarrow{\mu(\omega(S))} f_b \xrightarrow{\mu(\omega(S))} \ldots f_x \xrightarrow{\mu(\omega(S))} \ldots \tag{5}$$

Such a behavior is called a *Fire-and-Exit* behavior. It is quite different from a threaded approach in that all synchronization points are explicit. The role of the function f_{init} in (5) is crucial. This function has to provide the starting current function state $c_{init} \in C$

$$c_{init} = f_{init}(channel_a \ldots channel_z). \tag{6}$$

It evaluates first and only once, and it may read tokens from one or more external ports to return the starting current function state.

The SBF model is reminiscent of the *Applicative State Transition* (AST) node in the systolic array model of computation [34] introduced by Annevelink [1], after the *Applicative State Transition* programming model proposed by Backus [2]. The AST node, like the SBF actor, comprises a function repertoire, a selection function μ, and a transition function ω. However, the AST node does not have a private memory, and the initial function state c_{init} is read from a unique external channel. Moreover, AST nodes communicate through register channels. The SBF actor model is also reminiscent of the CAL, which is discussed in Sect. 3.

Clearly, the actors in decidable dataflow models [31] and the multidimensional dataflow models [37], and the processes in polyhedral precess networks [73] and Kahn process networks [21] are special cases of the SBF actor. Analysis is still possible in case the SBF-based DFAN originates from *weakly dynamic* nested loop programs, see Sect. 7.1, `Dynloop` programs, see Sect. 7.2, WLAP programs, see Sect. 7.3, and some parameterized polyhedral process networks, see Sect. 7.4.

Actors in dataflow actor networks communicate point to point over FIFO buffered channels. Conceptually, buffer capacities are unlimited, and actors synchronize by means of a *blocking read* protocol: an actor that attempts to read tokens from a specific channel will block whenever that channel is empty. Writing tokens will always proceed. Of course, channel FIFO capacities are not unlimited in practice, so that a mapped DFAN does have a *blocking write* protocol as well: an actor that attempts to write tokens to a specific channel will block whenever that channel is full. A function $[c,d] = f(a,b)$ has to bind to input ports p_x and p_y, respectively, when the function has to receive its arguments a and b, in this order. Similarly, that function has to bind to ports q_x and q_y, respectively, when ports q_x and q_y are to receive results c and d, in this order.

In the SBF actor, the controller's selection function μ selects both a function from the function repertoire and the corresponding input and output ports. Note that a function that is selected from the function repertoire may read arguments and write results from non-unique input ports and to non-unique output ports, respectively, as is also the case with polyhedral process networks [73].

This allows to separate function selection and binding, so that reading, executing, and writing can proceed in a pipelined fashion. Although standard blocking read and blocking write synchronization is possible, the SBF actor allows for a more general deterministic dynamic approach. In this approach, the actor behavior is divided into a *channel checking* part and a *scheduling* part. See also Sect. 5. In the channel checking part, channels are visited *without empty or full channel blocking*. A visited input channel C_{in} returns a $C_{in}.1$ signal when the channel is not empty, and a $C_{in}.0$ signal when the channel is empty. And similarly for output channels. These signals indicate whether or not a particular function from the function repertoire can fire. In the scheduling part, a function can only be invoked when the channel checking signals allow it to fire. If not, then the function will not be invoked. As a consequence, the actor is blocked. Clearly channel checking and scheduling can proceed in parallel.

9 Summary

This chapter, has reviewed several DSP-oriented dataflow models of computation that are oriented towards representing dynamic dataflow behavior. As signal processing systems are developed and deployed for more complex applications, exploration of such generalized dataflow modeling techniques is of increasing importance. This chapter has complemented the discussion in [31], which focuses on the relatively mature class of decidable dataflow modeling techniques, and builds on the dynamic dataflow principles introduced in certain specific forms [15, 21].

Acknowledgements In this work, Bhattacharyya has been supported in part by the US Air Force Office of Scientific Research. The authors also thank Marc Geilen (m.c.w.geilen@tue.nl) and Sander Stuijk (s.stuijk@tue.nl), both from the Eindhoven University of Technology, for their contribution to Sect. 6.

References

1. Annevelink, J.: HIFI: A design method for implementing signal processing algorithms on VLSI processor arrays. Ph.D. thesis, Delft University of Technology, Department of Electical Engineering, Delft, The Netherlands (1988)
2. Backus, J.: 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 (1978)

3. Bekooij, M., Hoes, R., Moreira, O., Poplavko, P., Pastrnak, M., Mesman, B., Mol, J., Stuijk, S., Gheorghita, V., van Meerbergen, J.: Dataflow analysis for real-time embedded multiprocessor system design. In: P. van der Stok (ed.) Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, pp. 81–108. Springer (2005)
4. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Proc. International Conference on Compiler Construction (ETAPS CC'10). Paphos, Cyprus (2010)
5. Berg, H., Brunelli, C., Lucking, U.: Analyzing models of computation for software defined radio applications. In: Proceedings of the International Symposium on System-on-Chip (2008)
6. Bhattacharya, B., Bhattacharyya, S.S.: Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing **49**(10), 2408–2421 (2001)
7. Bhattacharyya, S.S., Buck, J.T., Ha, S., Lee, E.A.: Generating compact code from dataflow specifications of multirate signal processing algorithms. IEEE Transactions on Circuits and Systems — I: Fundamental Theory and Applications **42**(3), 138–150 (1995)
8. Bhattacharyya, S.S., Eker, J., Janneck, J.W., Lucarz, C., Mattavelli, M., Raulet, M.: Overview of the MPEG reconfigurable video coding framework. Journal of Signal Processing Systems (2010). DOI:10.1007/s11265-009-0399-3
9. Bhattacharyya, S.S., Leupers, R., Marwedel, P.: Software synthesis and code generation for DSP. IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing **47**(9), 849–875 (2000)
10. Bijlsma, T., Bekooij, M.J.G., Smit, G.J.M.: Inter-task communication via overlapping read and write windows for deadlock-free execution of cyclic task graphs. In: Proceedings SAMOS'09, pp. 140–148. Samos, Greece (2009)
11. Buck, J.T.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley (1993)
12. Collard, J.F.: Automatic parallelization of while-loops using speculative execution. Int. J. Parallel Program. **23**(2), 191–219 (1995)
13. Deprettere, E.F., Rijpkema, E., Kienhuis, B.: Translating imperative affine nested loop programs to process networks. In: E.F. Deprettere, J. Teich, S. Vassiliadis (eds.) Embedded Processor Design Challenges, LNCS 2268, pp. 89–111. Springer, Berlin (2002)
14. Eker, J., Janneck, J.W.: CAL language report, language version 1.0 — document edition 1. Tech. Rep. UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley (2003)
15. Falk, J., Haubelt, C., Zebelein, C., Teich, J.: Integrated modeling using finite state machines and dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
16. Feautrier, P.: Dataflow analysis of scalar and array references. Int. Journal of Parallel Programming **20**(1), 23–53 (1991)
17. Feautrier, P.: Automatic parallelization in the polytope model. In: The Data Parallel Programming Model, pp. 79–103 (1996)
18. Feautrier, P., Collard, J.F.: Fuzzy array dataflow analysis. Tech. rep., Ecole Normale Superieure de Lyon (1994). ENS-Lyon/LIP N^o 94-21
19. Gao, G.R., Govindarajan, R., Panangaden, P.: Well-behaved programs for DSP computation. In: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (1992)
20. Geilen, M.: Synchronous dataflow scenarios. ACM Trans. Embed. Comput. Syst. **10**(2), 16:1–16:31 (2011)
21. Geilen, M., Basten, T.: Kahn process networks and a reactive extension. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
22. Geilen, M.C.W., Basten, T., Theelen, B.D., Otten, R.J.H.M.: An algebra of pareto points. Fundamenta Informaticae **78**(1), 35–74 (2007)

23. Geilen, M.C.W., Falk, J., Haubelt, C., Basten, T., Theelen, B.D., Stuijk, S.: Performance analysis of weakly-consistent scenario-aware dataflow graphs. Tech. Rep. ESR-2011-03, Eindhoven University of Technology (2011)
24. Geilen, M., Stuijk, S.: Worst-case performance analysis of synchronous dataflow scenarios. In: Proceedings of the eighth IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, CODES/ISSS '10, pp. 125–134. ACM, New York, NY, USA (2010)
25. Geuns, S., Bijlsma, T., Corporaal, H., Bekooij, M.: Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems. In: Proc. Int. Conf. Design, Automation and Test in Europe (DATE'11). Grenoble, France (2011)
26. Gheorghita, S.V., Stuijk, S., Basten, T., Corporaal, H.: Automatic scenario detection for improved WCET estimation. In: Proceedings of the 42nd annual Design Automation Conference, DAC '05, pp. 101–104. ACM, New York, NY, USA (2005)
27. Girault, A., Lee, B., Lee, E.: Hierarchical finite state machines with multiple concurrency models. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **18**(6), 742 –760 (1999)
28. Griebl, M., Collard, J.F.: Generation of Synchronous Code for Automatic Parallelization of while-loops. EURO-PAR'95, Springer-Verlag LNCS, number 966, pp. 315–326 (1995)
29. Griebl, M., Lengauer, C.: A communication scheme for the distributed execution of loop nests with while loops. Int. J. Parallel Programming **23** (1995)
30. Gu, R., Janneck, J., Raulet, M., Bhattacharyya, S.S.: Exploiting statically schedulable regions in dataflow programs. In: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, pp. 565–568. Taipei, Taiwan (2009)
31. Ha, S., Oh, H.: Decidable dataflow models for signal processing: Synchronous dataflow and its extensions. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
32. Haykin, S.: Adaptive Filter Theory. Prentice Hall (1996)
33. Hermanns, H.: Interactive Markov chains: and the quest for quantified quality. Springer-Verlag, Berlin, Heidelberg (2002)
34. Hu, Y.H., Kung, S.Y.: Systolic arrays. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
35. Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. of Information Processing (1974)
36. Kee, H., Wong, I., Rao, Y., Bhattacharyya, S.S.: FPGA-based design and implementation of the 3GPP-LTE physical layer using parameterized synchronous dataflow techniques. In: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, pp. 1510–1513. Dallas, Texas (2010)
37. Keinert, J., Deprettere, E.F.: Multidimensional dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
38. Kienhuis, B., Deprettere, E.F.: Modeling stream-based applications using the SBF model of computation. Journal of Signal Processing Systems **34**(3), 291–299 (2003)
39. Kienhuis, B., Rijpkema, E., Deprettere, E.F.: Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In: Proc. 8th International Workshop on Hardware/Software Codesign (CODES'2000). San Diego, CA, USA (2000)
40. Ko, M., Zissulescu, C., Puthenpurayil, S., Bhattacharyya, S.S., Kienhuis, B., Deprettere, E.: Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. IEEE Transactions on Signal Processing **55**(6), 3126–3138 (2007)
41. Lin, Y., Choi, Y., Mahlke, S., Mudge, T., Chakrabarti, C.: A parameterized dataflow language extension for embedded streaming systems. In: Proceedings of the International Symposium on Systems, Architectures, Modeling and Simulation, pp. 10–17 (2008)
42. Mattavelli, M., Raulet, M., Janneck, J.W.: MPEG reconfigurable video coding. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)

43. Moreira, O.: Temporal analysis and scheduling of hard real-time radios running on a multi-processor. Ph.D. thesis, Eindhoven University of Technology (2012)
44. Nadezhkin, D., Nikolov, H., Stefanov, T.: Translating affine nested-loop programs with dynamic loop bounds into polyhedral process networks. In: ESTImedia, pp. 21–30 (2010)
45. Nadezhkin, D., Stefanov, T.: Automatic derivation of polyhedral process networks from while-loop affine programs. In: ESTImedia, pp. 102–111 (2011)
46. Neuendorffer, S., Lee, E.: Hierarchical reconfiguration of dataflow models. In: Proceedings of the International Conference on Formal Methods and Models for Codesign (2004)
47. Neuendorffer, S., Lee, E.: Hierarchical reconfiguration of dataflow models. In: Proc. of MEMOCODE, pp. 179–188 (2004)
48. Nikolov, H., Stefanov, T., Deprettere, E.: Systematic and automated multi-processor system design, programming, and implementation. IEEE Transactions on Computer-Aided Design **27**(3), 542–555 (2008)
49. Plishker, W., Sane, N., Bhattacharyya, S.S.: A generalized scheduling approach for dynamic dataflow applications. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, pp. 111–116. Nice, France (2009)
50. Plishker, W., Sane, N., Bhattacharyya, S.S.: Mode grouping for more effective generalized scheduling of dynamic dataflow applications. In: Proceedings of the Design Automation Conference, pp. 923–926. San Francisco (2009)
51. Plishker, W., Sane, N., Kiemb, M., Anand, K., Bhattacharyya, S.S.: Functional DIF for rapid prototyping. In: Proceedings of the International Symposium on Rapid System Prototyping, pp. 17–23. Monterey, California (2008)
52. Poplavko, P., Basten, T., van Meerbergen, J.L.: Execution-time prediction for dynamic streaming applications with task-level parallelism. In: DSD, pp. 228–235 (2007)
53. Raman, E., Ottoni, G., Raman, A., Bridges, M.J., August, D.I.: Parallel-stage decoupled software pipelining. In: Proc. 6th annual IEEE/ACM international symposium on Code generation and optimization, pp. 114–123 (2008)
54. Rauchwerger, L., Padua, D.: Parallelizing while loops for multiprocessor systems. In: In Proceedings of the 9th International Parallel Processing Symposium (1995)
55. Rijpkema, E., Deprettere, E., Kienhuis, B.: Deriving process networks from nested loop algorithms. Parallel Processing Letters **10**(2), 165–176 (2000)
56. Roquier, G., Wipliez, M., Raulet, M., Janneck, J.W., Miller, I.D., Parlour, D.B.: Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In: Proceedings of the IEEE Workshop on Signal Processing Systems (2008)
57. Saha, S., Puthenpurayil, S., Bhattacharyya, S.S.: Dataflow transformations in high-level DSP system design. In: Proceedings of the International Symposium on System-on-Chip, pp. 131–136. Tampere, Finland (2006)
58. Shlien, S.: Guide to MPEG-1 audio standard. Broadcasting, IEEE Transactions on **40**(4), 206 –218 (1994)
59. Shojaei, H., Ghamarian, A., Basten, T., Geilen, M., Stuijk, S., Hoes, R.: A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for CMP run-time management. In: Design Automation Conf., DAC 09, Proc., pp. 917–922. ACM (2009)
60. Stefanov, T., Deprettere, E.: Deriving process networks from weakly dynamic applications in system-level design. In: Proc. IEEE-ACM-IFIP International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS'03), pp. 90–96. Newport Beach, California, USA (2003)
61. Stefanov, T., Kienhuis, B., Deprettere, E.: Algorithmic transformation techniques for efficient exploration of alternative application instances. In: Proc. 10th Int. Symposium on Hardware/-Software Codesign (CODES'02), pp. 7–12. Estes Park CO, USA (2002)
62. Stuijk, S., Geilen, M., Basten, T.: SDF^3: SDF For Free. In: Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings, pp. 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA (2006). DOI 10.1109/ACSD.2006.23. URL http://www.es.ele.tue.nl/sdf3

63. Stuijk, S., Geilen, M., Basten, T.: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. IEEE Trans. on Computers **57**(10), 1331–1345 (2008)
64. Stuijk, S., Geilen, M., Basten, T.: A predictable multiprocessor design flow for streaming applications with dynamic behaviour. In: Proceedings of the Conference on Digital System Design, DSD '10, pp. 548–555. IEEE (2010). DOI 10.1109/DSD.2010.31
65. Stuijk, S., Geilen, M.C.W., Theelen, B.D., Basten, T.: Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In: ICSAMOS, pp. 404–411 (2011)
66. Theelen, B.D., Geilen, M.C.W., Stuijk, S., Gheorghita, S.V., Basten, T., Voeten, J.P.M., Ghamarian, A.: Scenario-aware dataflow. Tech. Rep. ESR-2008-08, Eindhoven University of Technology (2008)
67. Theelen, B.D., Geilen, M.C.W., Voeten, J.P.M.: Performance model checking scenario-aware dataflow. In: Proceedings of the 9th international conference on Formal modeling and analysis of timed systems, FORMATS'11, pp. 43–59. Springer-Verlag, Berlin, Heidelberg (2011)
68. Theelen, B.D.: A performance analysis tool for scenario-aware streaming applications. In: QEST, pp. 269–270 (2007)
69. Theelen, B.D., Florescu, O., Geilen, M.C.W., Huang, J., van der Putten, P.H.A., Voeten, J.P.M.: Software/hardware engineering with the parallel object-oriented specification language. In: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE '07, pp. 139–148. IEEE Computer Society, Washington, DC, USA (2007)
70. Theelen, B.D., Geilen, M.C.W., Basten, T., Voeten, J.P.M., Gheorghita, S.V., Stuijk, S.: A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Proceedings of MEMOCODE, pp 185194, pp. 185–194. IEEE Computer Society Press (2006)
71. Theelen, B.D., Katoen, J.P., Wu, H.: Model checking of scenario-aware dataflow with CADP. In: DATE, pp. 653–658 (2012)
72. Turjan, A., Kienhuis, B., Deprettere, E.: Realizations of the Extended Linearization Model. in Domain-Specific Embedded Multiprocessors (Chapter 9), Marcel Dekker, Inc. (2003)
73. Verdoolaege, S.: Polyhedral process networks. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
74. Verdoolaege, S., Nikolov, H., Stefanov, T.: pn: a tool for improved derivation of process networks. EURASIP J. Embedded Syst. (2007)
75. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., Bruynooghe, M.: Counting integer points in parametric polytopes using Barvinok's rational functions. Algorithmica (2007)
76. Wiggers, M.: Aperiodic multiprocessor scheduling. Ph.D. thesis, University of Twente (2009)
77. Willink, E.D., Eker, J., Janneck, J.W.: Programming specifications in CAL. In: Proceedings of the OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture (2002)
78. Zhai, J.T., Nikolov, H., Stefanov, T.: Modeling adaptive streaming applications with parameterized polyhedral process networks. In: Proceedings of the 48th Design Automation Conference, DAC '11, pp. 116–121. ACM, New York, NY, USA (2011)