Dynamic Dataflow Graphs

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

Abstract Much of the work to date on dataflow models for signal processing system design has focused on decidable dataflow models. This chapter reviews more general dataflow modeling techniques targeted to applications that include dynamic dataflow behavior. The complexity in such applications demands for increased degrees of agility and flexibility in dataflow models. With the application of dataflow techiques addressing these challenges, interest in classes of more general dataflow models has risen correspondingly. 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. The dynamic dataflow models covered in this chapter are Boolean Dataflow, CAL, Parameterized Dataflow, Enable-Invoke Dataflow, Scenario-Aware Dataflow, and Dynamic Polyhedral Process Networks.

1 Motivation for Dynamic DSP-Oriented Dataflow Models

The decidable dataflow models covered in [30] 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 all of the functionality in terms of purely decidable dataflow representations. For ex-

Bart D. Theelen
Océ Technologies B.V., The Netherlands
e-mail: `bart.theelen@oce.com`

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

Shuvra S. Bhattacharyya
University of Maryland, USA, and
Tampere University of Technology, Finland
e-mail: `ssb@umd.edu`

ample, functionality that involves conditional execution of dataflow subsystems or actors with dynamically varying production and consumption rates can in general not be expressed with decidable dataflow models.

The need for expressive power beyond what decidable dataflow techniques provide is becoming increasingly important in the design and implementation of signal processing systems. This is due to the increasing levels of application dynamics that must be supported in such systems. Examples include 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 [18].

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), 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. However, in general, specifications may exist that "break" these analysis techniques in the sense that compile-time analysis gives inconclusive results.

Dynamic dataflow techniques can be divided into two general classes:

- Models formulated explicitly in terms of interacting combinations of state machines and dataflow graphs. In this case, the dataflow dynamics are represented directly in terms of transitions within one or more underlying state machines
- Models 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 are covered in [30] — 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.

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

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

to provide a representative cross-section of relevant dynamic dataflow techniques. The emphasis is on techniques for which useful forms of compile-time analysis methods 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* (SDF) 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*. Such control tokens 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. 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. The consumption rate on C_p is statically fixed at one token per invocation of A. 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 consumed from a control input port or of a token 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. The number of tokens produced on q during a given invocation is a two-valued function of the data value 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 Figure 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.

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, port s_c accepts Boolean valued tokens, and the production rate on s_o 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.

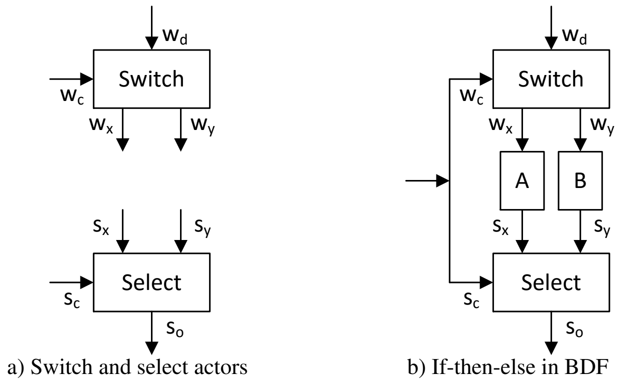


Fig. 1: (a) Switch and select actors in Boolean dataflow, and (b) An if-then-else construct expressed in terms of Boolean dataflow.

Switch and select actors can be integrated along with other actors in various ways to express different kinds of control constructs. For example, Figure 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 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 [9]. Buck also showed 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 to demonstrate Turing completeness for other kinds of dynamic dataflow models, such as the *Enable-Invoke Dataflow* (EIDF) model described in Section 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. It is furthermore 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 [13]. It 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 [56, 29, 78]). 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) [6].

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. However, the 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. This flexibility enables the modeling of dynamic dataflow in CAL.

A CAL actor A can be represented as a sequence of four elements $\sigma_0(A)$, $\Sigma(A)$, $\Gamma(A)$, $pri(A)$, where $\Sigma(A)$ represents the set of all possible values that the state of A can take. $\sigma_0(A) \in \Sigma(A)$ 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. Finally, $pri(A)$ 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 the capability for *specialized region detection* in CAL programs is the technique of deriving and exploiting so-called *Statically Schedulable Regions* (SSRs) [29]. 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.

A further discussion of CAL can be found in [43], which presents 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 the class of dataflow models of computations that have a well-defined concept of a graph *iteration* [4]. For example, SDF and *Cyclo-Static* SDF (CSDF), which are discussed in [30], and *Multi-Dimensional* SDF (MDSDF), which is discussed in [39], 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 to generalize a specialized dataflow model such as SDF, CSDF, or MDSDF, the specialized model is referred to as the *base model*. The resulting dynamically parameterizable form of the base model is referred to as *parameterized* X, where X denotes the base model. For example, when Parameterized Dataflow is applied to SDF as the base model, the resulting model of computation is called *Parameterized Synchronous Dataflow* (PSDF). 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 [5] for SDF graphs – can be replaced by *parameterized looped schedules* [4, 41]. In such parameterized schedules, loop iteration counts are represented as symbolic expressions in terms of variables whose values can be adapted dynamically through computations 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. Also edge attributes, like the delay of an edge or the data type associated with tokens transferred across the edge, can be parameterized. A final parameterization example are graph attributes 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*, *subinit*, and *init* graphs. The body graph typically represents the functional "core" of the 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 and general definition for a body graph iteration would be a single SDF *iteration* of the body graph (as defined by the SDF repetitions vector [30]).

An iteration of the body graph can however also be defined as some constant number of iterations, e.g., 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. 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 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 called 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 [4]. 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 representing 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

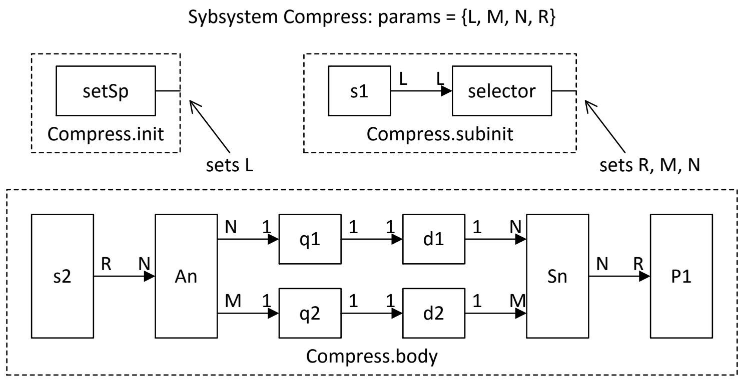


Fig. 2: An illustration of a speech compression system that is modeled using PSDF semantics. This illustration is adapted from [4].

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. The receiver side functionality is modeled in the body graph starting with the actors *d1* and *d2*, which represent dequantizers. The actor *Sn* (“synthesize”) reconstructs speech instances using corresponding blocks of AR coefficients and error signals. 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.

The model of Figure 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). In PCSDF, Parameterized Dataflow is integrated with CSDF as the base model instead of SDF.

Further details on the examplified speech compression application and its representations in PSDF and PCSDF, the semantics of Parameterized Dataflow and PSDF, and quasi-static scheduling techniques for PSDF can be found in [4]. Parameterized Cyclo-Static Dataflow (PCSDF), the integration of Parameterized Dataflow metamodeling with Cyclo-Static Dataflow, is explored further in [57]. The exploration of different models of computation, including PSDF and PCSDF, for modeling software defined radio systems is described in [3]. In [38], 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 described in [42]. General techniques for analysis and verification of hierarchically reconfigurable dataflow graphs are explored in [47].

5 Enable-Invoke Dataflow

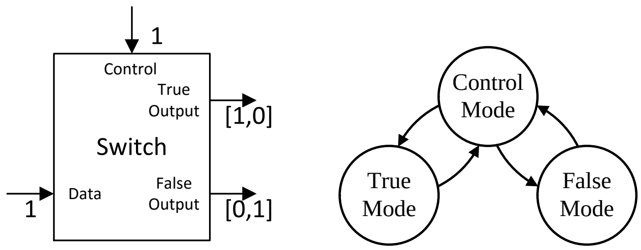
Enable-Invoke Dataflow (EIDF) is another DSP-oriented dynamic dataflow modeling technique [51]. The applicability of EIDF has been demonstrated in the context of behavioral simulation, FPGA implementation, and prototyping of different scheduling strategies [51, 49, 50]. This latter capability – prototyping 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 key implementation metrics [7]. 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*. Although each mode has constant production/consumption rate behavior, different modes can have different dataflow rates. Unlike other forms of mode-oriented dataflow specification, SDF-integrated starcharts [30], SysteMoc [14], and CAL (see Section 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 it is completely separated from mode execution. Furthermore, the approach improves 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, 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. The invoke function 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 (i.e., it 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.



a) Switch actor b) Mode transition diagram

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

c) Production and consumption behavior of modes

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

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*). DIF is a textual language for DSP-oriented dataflow graphs and Java-based specifications of intra-actor functionality covering enable and invoke functions, and next mode computations [51].

Figure 3 and Figure 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 Section 2.

6 Scenario-Aware Dataflow

This Section discusses *Scenario-Aware Dataflow* (SADF), which is a generalization of dataflow models with strict periodic or static behavior. Like many 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. 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* [25]. The scenarios commonly coincide with dissimilar (but within themselves more static) modes of opera-

```
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);
}
```

a) Constructor
(defines modes and dataflow behavior)

```
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;
}
```

b) Enable Function
(determines whether firing condition is met)

```
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;
    }
}
```

c) Invoke Function
(performs action and determines next mode)

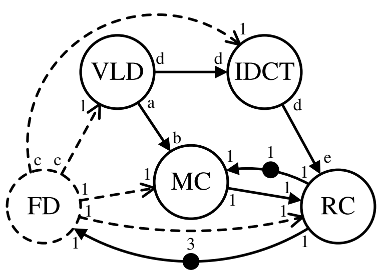
Fig. 4: An implementation of the `switch` actor design of Figure 3 in the functional DIF environment.

tion 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 [25]. The scenario-concept in SADF allows for more precise (quantitative) analysis results compared to applying SDF-based analysis techniques. Moreover, common subclasses of SADF can be synthesized into efficient implementations [66, 35].

6.1 SADF Graphs

We introduce SADF by some examples from the multi-media domain. We first consider the MPEG-4 video decoder for the Simple Profile from [71, 67]. 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 9 different scenarios are identified. SADF distinguishes two types of actors: *kernels* (solid vertices) model the data processing parts, whereas *detectors* (dashed vertices) 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



a) Actors and Channels

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

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

b) Parameterized Rates

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

c) Worst-Case Execution Times

Fig. 5: Modeling the MPEG-4 decoder with SADF.

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

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 Figure 5b. A value of 0 expresses that data dependencies are absent or that certain operations are not performed in those scenarios. Studying Figure 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 [71], which means that scenario changes as prescribed by the behavior of FD may only occur at iteration boundaries of each scenario graph. This is not necessarily true for SADF in general as discussed below.

SADF specifies execution times of actors (from a selected time domain, see Subsection 6.2) per scenario. Figure 5c lists the worst-case execution times of the MPEG-4 decoder for an ARM7TDMI processor. The tables in Figure 5 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 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 Subsection 6.2. The dynamics of the MPEG-4 decoder can be captured by a state-machine of 9 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 (see Section 2) and to EIDF (see Section 5). 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. We use *subscenario* to denote 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 sce-

[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.

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

We now turn our attention to the MP3 audio decoder example from [67] depicted in Figure 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 1152 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 types as 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.

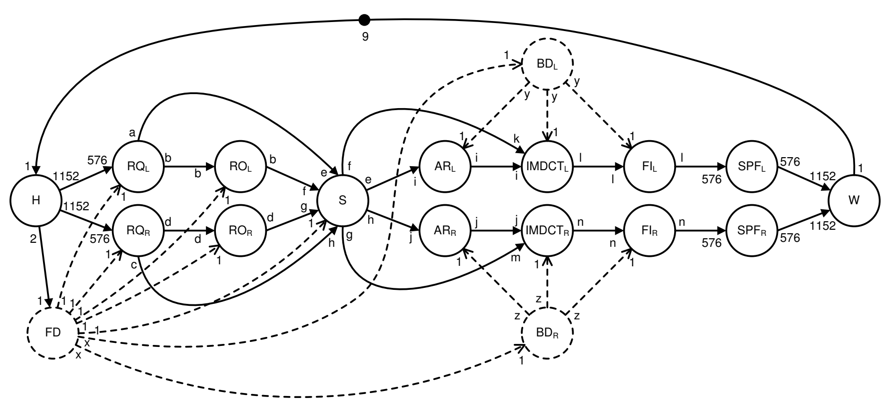


Fig. 6: Modeling an MP3 decoder with SADF using hierarchical control.

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

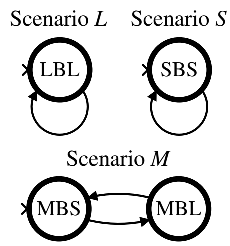
Rate	Scenario L	S	M
a, c	576	0	36
b, d	0	576	540

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

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

a) Parameterized Rates



b) State Machines for BD_L and BD_R

Fig. 7: Properties of the MP3 decoder model.

Figure 7a lists the parameterized rates for the MP3 decoder. Only five combinations of frame types occur for the two audio channels combined. We use a two-letter abbreviation to indicate the combined fame type for the left and right audio channel respectively: *LL*, *SS*, *LS* and *SL*. Mixed frames *M* cover both audio channels simultaneously. Detector FD determines the frame type with a state machine of 5 states, each uniquely identifying 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 RQ_L, RO_L and RQ_R, 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*.

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 $\{\mathrm{LBL}, \mathrm{SBS}, \mathrm{MBL}, \mathrm{MBS}\}$. 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 4 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 Figure 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 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 Heterogeneous Dataflow [26]. 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). We briefly discuss consistency of SADF graphs. 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 (although still pipelined). Such SADF graphs are said to be *strongly consistent* [71], 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* [67, 22]. 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 typically enables worst/best-case analysis. Applying the techniques in [19, 22, 23] 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 also 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 [68] compute that the MPEG-4 decoder processes on average 0.426 frames per kCycle.

The semantics of SADF graphs where Markov chains are associated to detectors while assuming generic discrete execution time distributions[5] has been defined in [67] by using *Timed Probabilistic Systems* (TPS) as formal sematic model. Such transition systems operationalize the behavior with states and guarded transitions

[5] This covers the case of constant execution times as so-called point distributions [67, 68].

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, Markov Decision Processes, and Markov Chains by defining reward structures as also used in (probabilistic or quantitative) model checking. In [68], for example, specific properties of dataflow models in general and SADF in particular are discussed that enable substantial state-space reductions during such analysis. The underlying techniques have been implemented in [69] in the SDF^3 tool kit [63], covering the computation of worst/best-case and average-case properties for SADF including throughput and various forms of latency and buffer occupancy metrics [69].

Other variants of Scenario-Aware Dataflow have been proposed that are supported by exact analysis techniques using formal sematic models. The techniques presented in [72, 36, 37] exploit *Interactive Marchov Chains* (IMC) to combine the association of Markov chains to detectors with exponentially distributed execution times, which allows for instance computing the response time distribution of the MPEG-4 decoder to complete processing the first frame [72]. A further generalisation of the time model for Scenario-Aware Dataflow with Markov chains associated to detectors is proposed in [31]. This generalisation is based on the formal sematic model of *Stochastic Timed Automata* (STA) and allows for scenario-dependent cost annotations to compute for instance energy consumption.

When abstracting 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* [44, 77, 19]. This property 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 SADF graphs with a single detector (often 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 explicitly exploring the underlying state-space. Instead, the analysis is performed by means of state-space analysis and maximum-cycle ratio analysis of the equivalent but much smaller $(\max,+)$-automaton [19, 22, 23]. Reference [22] shows how this analysis can be extended for weakly-consistent SADF graphs. An alternative to using $(\max,+)$-algebra is proposed in [60], where the formal semantic model of *Timed Automata* (TA) is exploited to enable analyzing various qualitative and quantitative properties.

In case exact computation is hampered by state-space explosion, [71, 69] exploit an automated translation into process algebraic models expressed in the *Parallel Object-Oriented Specification Language* (POOSL) [70], which supports statistical model checking (simulation-based estimation) of various average-case properties.

6.3 Synthesis

FSM-based SADF graphs have been extensively studied for implementation on (heterogeneous) multi-processor platforms [65, 35]. 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 [63] 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].

We highlight two key aspects of the design flow of [65, 63]. 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 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 [65] 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 [64] 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 [21] 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 [65] 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 [65] 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 [1], it is shown that

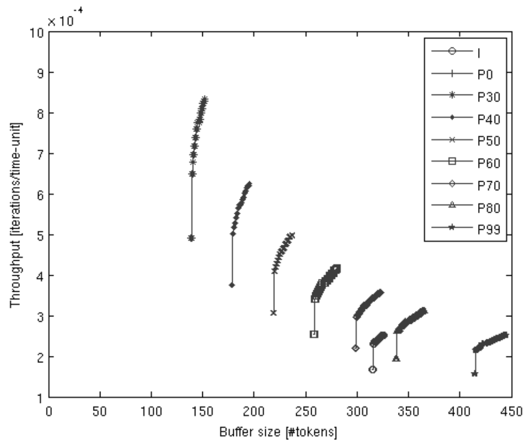


Fig. 8: Throughput/buffer size trade-off space for the MPEG-4 decoder.

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) [74] 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 [61],
2. *loops* with no condition on the bounds [45],
3. *while* statements other than `while(1)` [46],
4. dynamic parameters [79].

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 DANLP.

[6] The corresponding tool is called PNgen [75], and is part of the `Daedalus` design framework [48], `http://daedalus.liacs.nl`.

7.1 Weakly Dynamic Programs

Whereas condition statements in an SANLP 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 [61]. A simple example of a WDP is shown in Figure 9. The question here is whether the argument of function $F3$ originates from the output of function $F2$ or function $F1$.

In the case of an SANLP, the input-output equivalent PPN is obtained by:

1. Converting the SANLP – by means of an *array analysis* [15, 16] – into a *Single Assignment Code* (SAC) used in the compiler community and the systolic array community [33]
2. Deriving from the SAC a *Polyhedral Reduced Dependence Graph* (PRDG) [55]
3. Constructing the PPN from the PRDG [40, 55, 11]

While in an 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 at compile time whether or not they will be read or written. For a WDP not all dependences are known at compile time and therefore, the analysis must be based on the so-called *Fuzzy Array Dataflow Analysis* (FADA) [17]. This approach allows the conversion of a WDP to a dSAC. The procedure to generate the dSAC is out of the scope. The dSAC for the WDP in Figure 9 is shown in Figure 10.

Parameter `C` in the dSAC of Figure 10 is emerging from the if-statement in line 8 of the original program shown in Figure 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 Figure 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 [61] 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* [55, 11]. They contain all information related to the data dependencies between functions in

```
1  %parameter N 8 16;
2
3  for i = 1:1:N,
4   [x(i), t(i)] = F1(...);
5  end
6
7  for i = 1:1:N,
8   if  t(i) <= 0,
9    [x(i)] = F2( x(i) );
10  end
11  [...] = F3( x(i) );
12 end
```

Fig. 9: Pseudo code of a simple Weakly Dynamic Program.

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* (LBS), 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. An 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 Figure 10. The exact iterations i are not known at compile time because of the dynamic condition at line 14 in the dSAC. 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

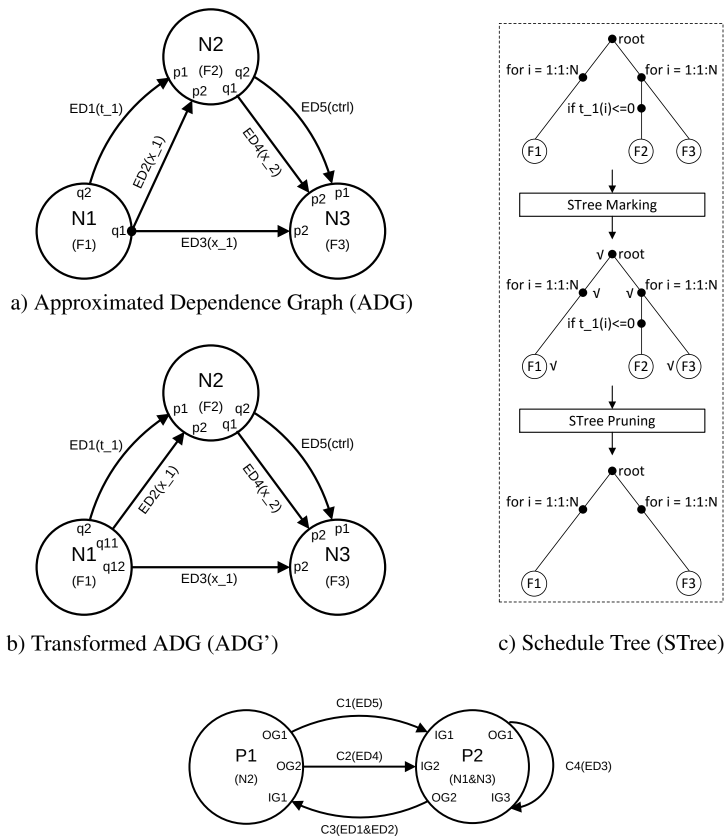
```
1  %parameter N 8 16;
2
3  for i = 1:1:N,
4    ctrl(i) = N+1;
5  end
6  for i = 1:1:N,
7   [out_0, out_1] = F1(...);
8   [x_1(i)] = opd(out_0);
9   [t_1(i)] = opd(out_1);
10 end
11
12 for i = 1:1:N,
13  [t_1(i)] = ipd(t_1(i));
14  if t_1(i) <= 0,
15   [in_0] = ipd(x_1(i));
16   [out_0] = F2(in_0);
17   [x_2(i)] = opd(out_0);
18   [ ctrl(i) ] = opd( i );
19  end
20
21  C = ipd( ctrl(i) );
22  if i = C,
23   [in_0] = ipd(x_2(C));
24  else
25   [in_0] = ipd(x_1(i));
26  end
27
28  [out_0] = F3(in_0);
29  [...] = opd(out_0);
30 end
```

Fig. 10: Dynamic Single Assignment Code for the example if Figure 9.

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 [62]. The STree 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, i.e., $F1$, $F2$, and $F3$ in Figure 10 [61]. Figure 11 depicts a summary.

The difference between the ADG in Figure 11a and the transformed ADG in Figure 11b is that an ADG may have several input ports connected to a single output



a) Approximated Dependence Graph (ADG)

b) Transformed ADG (ADG')

c) Schedule Tree (STree)

d) Process Network (PN) model

Fig. 11: Examples of a) Approximated Dependence Graph (ADG) model; b) Transformed ADG; c) Schedule Tree and Transformations; d) Process Network model.

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 [34]). Parsing the STree in Figure 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 Figure 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 Figure 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 Figure 11c. See [61] for details.

In a (static) PPN, there are two models of FIFO communication [73], 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 [61].

Buffer sizes can be determined using the procedure given in [74, 75]. It however needs a conservative strategy (i.e., an over-estimation) due to the fact that the rate and the exact amount of data tokens transferred over a particular data channel is unknown at compile-time. Such over-estimation can be achieved by modifying the iteration domains of all input/output ports, such that all dynamic if-conditions defining any of these iteration domains always evaluate to `true`.

7.2 Dynamic Loop-Bounds

While loop bounds in an SANLP 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 [45]. A simple example of a `Dynloop` program is shown in Figure 12a.

A `Dynloop` program can be cast in the form of a WDP, see Subsection 7.1. The WDP corresponding to the `Dynloop` program in Figure 12a is shown in Figure 12b. The maximum value of $f()$, denoted by `max_f` in line 5 in Figure 12b is substituted for the upper bound of the loop at line 4 in Figure 12a. The value of `max_f` can be determined by studying the range of function $f()$[7]. As in Subsection 7.1, a dSAC can now be obtained by means of a FADA [17]. 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

[7] If that is not possible, then an alternative way to estimate `max_f` is given in [45].

```
%parameter N 1 10;

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

a) 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] )
```

b) Equivalent Weakly Dynamic Program

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

as done in Subsection 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 [45].

The final dSAC is shown in Figure 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:

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

```
%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

For brevity, the initialization is not shown in Figure 13. After applying the standard *linearization* [73], and its extension of Subsection 7.1, and estimating buffer sizes as also described Subsection 7.1, the resulting PPN is as shown in Figure 14.

7.3 Dynamic While-Loops

While only `while(1)` loops are allowed in an SANLP program, in a DANLP program any while-loop is acceptable. Such DANLP programs have been called *While-Loop Affine Programs* (WLAP) in [46]. There are a number of publications that address the problem of while loops parallelization [53, 54, 10, 27, 28, 8, 24, 2]. 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 with distributed memory,
- provides an automatic data-dependence analysis procedure,
- exposes and utilizes all available parallelism.

An example is shown in Figure 15a. The question is again from where for example function $F7$ gets its scalar argument x. Because this is not known at compile-time, a FADA analysis [17] is necessary to find all data dependencies. The approach to convert a WLAP program to an input-output equivalent PPN goes in four steps:

1. All data-dependency relations in the initial WLAP program have to be found by applying 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.
2. Based on the results of the analysis, the initial WLAP is transformed into a dSAC representation, see Subsection 7.1. Parameters that are introduced by the FADA appear in the dSAC, and their values are assigned using control variables.

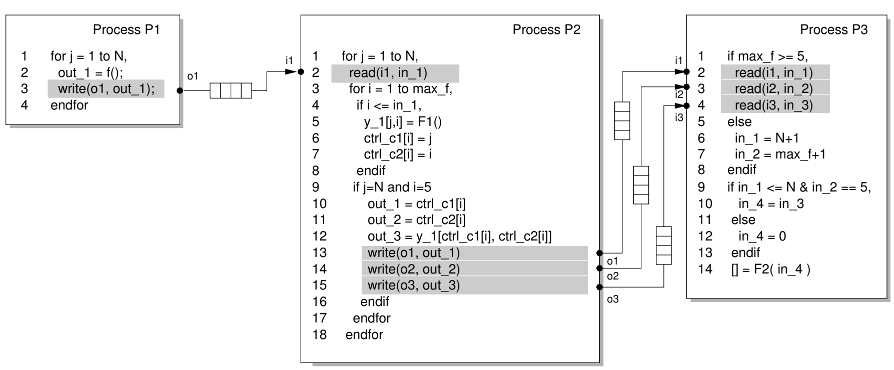


Fig. 14: The final PPN derived from the program in Figure 13.

3. The control variables are generated in a way that extends the methods in Subsection 7.1 and Subsection 7.2 to be applicable for WLAP programs as well [46].
4. The topology of the corresponding PPN is derived as well as the code to be executed in the processes of the PPN.

The result of step 2 for the example in Figure 15a is shown in Figure 15b. 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. It follows from line 23 in Figure 15b that control variable `ctrl_x_5` is initialized to `ctrl_x_5 = (N+1,0)` at line 3 in Figure 15b.

Where does control variable `ctrl_x_5` come from? It comes from the construction of the dSAC. The procedure to derive the dSAC is largely based on [17] and its extension in Subsection 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,

```
&parameter EPS 0.005

for i = 1 to N,
  y[i] = F1()
  x = F2( y[i] )
 while ( x >= EPS )
   x = F3()
   for j = i+1 to N+1,
     y[j] = F4( y[j-1] )
     x = F5( x, y[j] )
   end
    y[i] = F6( x )
  end
   out = F7( x )
 end
```

a) 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 = σx(⟨W,(i,w)⟩) >= EPS),
9      w = w + 1
11     x_3[i,w] = F3()
11     for j = i+1 to N+1,
12       in_4 = σy(⟨S4,(i,w,j)⟩)
13       y_4[i,w,j] = F4( in_4 )
14       in_5_x = σx(⟨S5,(i,w,j)⟩)
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 = σx(⟨S6,(i,w)⟩)
20     y_6[i,w] = F6( in_6 )
21   end
22   ctrl_x_5[i] = ctrl_x_5
23   (α,β) = ctrl_x_5[i]
24   in_7 = σx(⟨S7,(i,α,β)⟩)
25   out = F7( in_7 )
26 end
```

b) The corresponding final dSAC

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

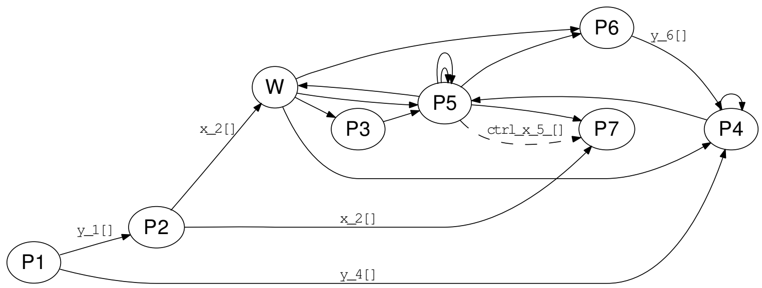


Fig. 16: The PPN for the program in Figure 15.

an additional control variable `ctrl_x_5_` is introduced right *after* the while-loop, see line 22 in Figure 15b. 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 Array Dataflow Analysis* (EADA) [15] 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.

Step 4 constructs the PPN from the dSAC. The PPN corresponding to the dSAC in Figure 15b is depicted in Figure 16. It consists of 8 processes and 18 channels. The processes $P1$–$P7$ correspond to the functions $F1$–$F7$ in Figure 15. Process W corresponds to the while condition at line 8 of the dSAC in Figure 15b.

The code for processes W, $P5$, and $P7$ is shown in Figure 17. Process W is an example of a process detecting the termination of the while-loop at line 5 in Figure 15a. 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 an SANLP program are static. In a DANLP, parameters can be dynamic. A *Polyhedral Process Network* (PPN) that is input-output equivalent to a DANLP program is called a *Parameterized Polyhedral Process Network*, which is abbreviated to P^{3}N. The formal definition of a P^{3}N is given in [79], and is only slightly different from the definition in [74]. Although the consistency of a P^{3}N has to be checked at run-time, still some analysis can be done at compile-time.

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.

An example P^{3}N is shown in Figure 18. Figure 18a is a static PPN of which process *P3* is shown in Figure 18b. Figure 18c presents a P^{3}N version of the PPN in

```
%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
```

a) Code of process W

```
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
```

b) Code of process P7

```
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
```

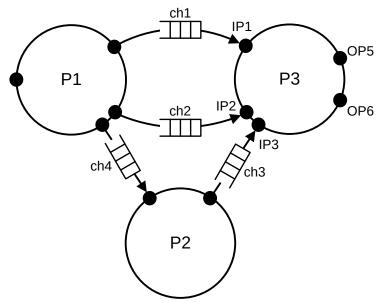
c) Code of process P5

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

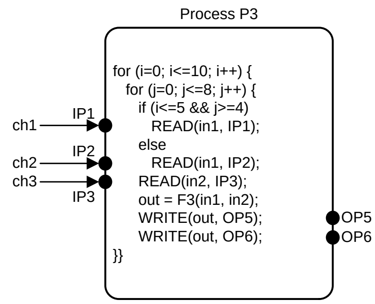
Figure 18a. Process $P3$ of the P^{3}N in Figure 18c is shown in Figure 18d. The PPN and the P^{3}N have the same *dataflow* topology. Processes *P2* and *P3* in the P^{3}N in Figure 18c are reconfigured by two parameters M and N whose values are updated from *the environment* at run-time using process *Ctrl* and FIFO channels *ch7*, *ch8*, and *ch9*. The P^{3}N in Figure 18c may be derived from a sequential program, yet it can also be constructed from library elements as in [30] or using the approach of [12].

Reference [74] explains that a parametric polyhedron $\mathcal{P}(\mathbf{p})$ is defined as $\mathcal{P}(\mathbf{p}) = \{(w, x_1, \ldots, x_d) \in \mathbb{Q}^{d+1} \mid A \cdot (w, x_1, \ldots, x_d)^T \geq B \cdot \mathbf{p} + b\}$ 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 $\mathcal{P}_{\mathbf{p}} = \{\mathbf{p} \in \mathbb{Q}^n \mid C \cdot \mathbf{p} \geq d\}$. The domain D_P of a process is defined as the set of all integral points in its underlying parametric polyhedron, i.e., $D_P = \mathcal{P}_P(\mathbf{p}) \cap \mathbb{Z}^{d+1}$. 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.

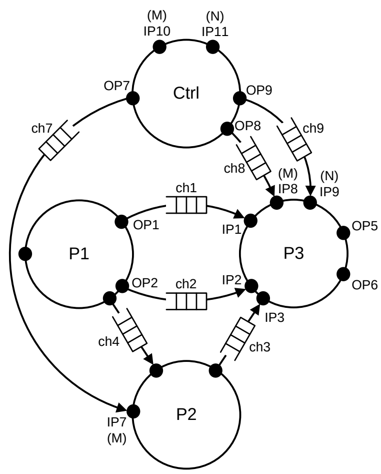
The following four notions play a role in the operational semantics of a P^{3}N:



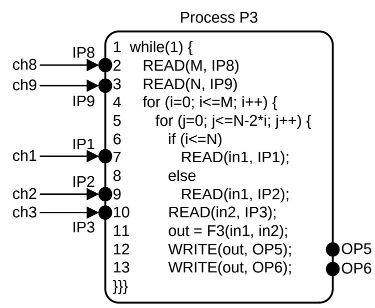
a) Example of a PPN



b) Process *P3* in the PPN in subfigure a)



c) Example Parameterized PPN



d) Process *P3* in the P^3N in subfigure c)

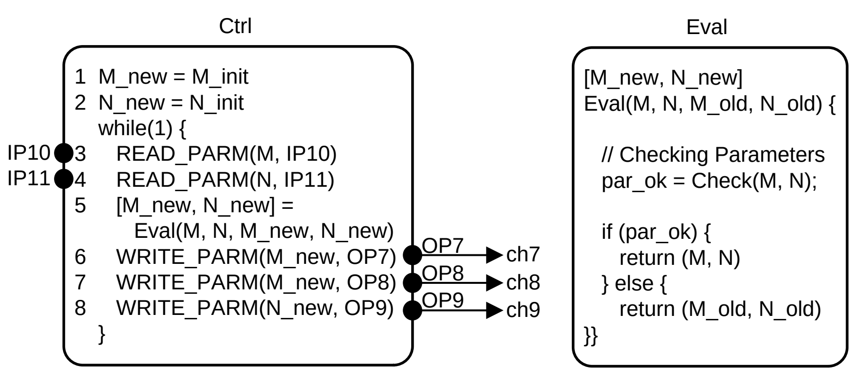
Fig. 18: (a) An example of a PPN, (b) process *P3* in the PPN, (c) an example of a P^3N, and (d) process *P3* in the P^3N.

- A **process iteration** of process P is a point $(w, x_1, ..., x_d) \in D_P$, where the following operations are performed sequentially: reading a token from each IP for which $(w, x_1, ..., x_d) \in D_{IP}$, executing process function F_P, and writing a token to each OP for which $(w, x_1, ..., x_d) \in D_{OP}$.
- A **process cycle** $CYC_P(\mathcal{S}, \mathbf{p}) \subset D_P$ is the set of lexicographically ordered points $\in D_P$ for a particular value of $w = \mathcal{S} \in \mathbb{Z}^+$. The lexical ordering is typically imposed by a loop nest.
- A **Process execution** E_P is a sequence of process cycles denoted by $CYC_P(1, \mathbf{p}_1) \rightarrow CYC_P(2, \mathbf{p}_2) \rightarrow ... \rightarrow CYC_P(k, \mathbf{p}_k)$, where $k \rightarrow \infty$.
- A point $Q_P(\mathcal{S}, \mathbf{p}_i) \in CYC_P(\mathcal{S}, \mathbf{p}_i)$ of process P is a **quiescent point** if $CYC_P(\mathcal{S}, \mathbf{p}_i) \in E_P$ and $\neg(\exists (w, x_1, ..., x_d) \in CYC_P(\mathcal{S}, \mathbf{p}_i) : (w, x_1, ..., x_d) \prec Q_P(\mathcal{S}, \mathbf{p}_{\mathcal{S}})$.

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 Figure 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 implementation of function `Eval` is given in Figure 19b. If the combination is valid, then function `Eval` returns the current parameter values (`M, N`). Otherwise, the last valid parameter combination (propagated through `M_new, N_new` in this example) is returned. After the evaluation of the parameter 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 consistency problem, consider channel *ch3* connecting processes *P2* and *P3* of the P^{3}N given in Figure 18c. The access of processes *P2* and *P3* to channel *ch3* is depicted in Figure 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



a) Process Ctrl   b) Process Function Eval

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

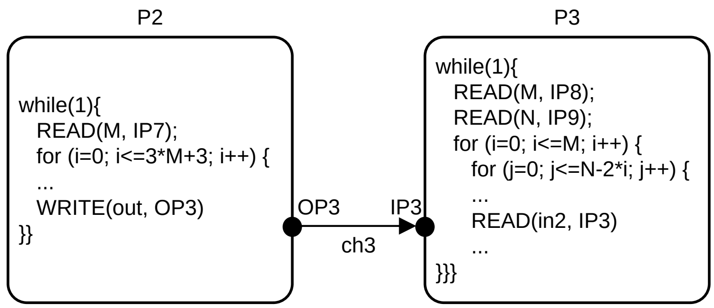


Fig. 20: Which combinations $(M,\ N)$ do ensure consistency of $\mathrm{P^3N}$?

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 $\mathrm{P^3N}$. 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 $\mathrm{P^3N}$.

Although the consistency of a $\mathrm{P^3N}$ 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 [76, 74]. In general, the number of points in domain $D_X = \mathcal{P}_X(\mathbf{p}) \cap \mathbb{Z}^{d+1}$, where X stand for either a process *P*, an input port *IP*, or an output port *OP*, is a set of quasi-polynomials [74].

For the example in Figure 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 $C1 = \{(M,N) \in \mathbb{Z}^2 \mid M \leq N \wedge 2M \geq 1+N\}$, $C2 = \{(M,N) \in \mathbb{Z}^2 \mid 2M \leq N\}$, and $\{0,1\}_N$ is a periodic coefficient with period 2^8. If in this example the range of the parameters is $0 \leq M,N \leq 100$, then there are only 10 valid parameter combinations. If $0 \leq M,N \leq 1000$, then there are 34 valid parameter combinations, and if $0 \leq M,N \leq 10000$, 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 de-

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

sign 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 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 [79] for further details.

8 Summary

This chapter reviewed several DSP-oriented dataflow models of computation that focus on 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 complemented the discussion in [30], which focuses on the relatively mature class of decidable dataflow modeling techniques, and builds on the dynamic dataflow principles introduced in certain specific forms [20, 14].

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 Section 6.

References

1. Bekooij, M., Hoes, R., Moreira, O., Poplavko, P., Pastrnak, M., Mesman, B., Mol, J.D., 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)
2. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: R. Gupta (ed.) Proceedings of the International Conference on Compiler Construction, CC, pp. 283–303. Springer (2010)
3. 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, SoC, pp. 1–4 (2008)
4. Bhattacharya, B., Bhattacharyya, S.S.: Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing **49**(10), 2408–2421 (2001)
5. 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)
6. 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)

7. 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)
8. 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 of the International Symposium on Systems, Architectures, Modeling, and Simulation, ICSAMOS, pp. 140–148 (2009)
9. 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)
10. Collard, J.F.: Automatic parallelization of while-loops using speculative execution. International Journal of Parallel Programming **23**(2), 191–219 (1995)
11. 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 (2002)
12. Desnos, K., Palumbo, F.: Dataflow modeling for reconfigurable signal processing systems. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
13. 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)
14. Falk, J., Neubauer, K., 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, third edn. Springer (2018)
15. Feautrier, P.: Dataflow analysis of scalar and array references. International Journal of Parallel Programming **20**(1), 23–53 (1991)
16. Feautrier, P.: Automatic parallelization in the polytope model. In: The Data Parallel Programming Model, pp. 79–103 (1996)
17. Feautrier, P., Collard, J.F.: Fuzzy array dataflow analysis. Tech. rep., Ecole Normale Superieure de Lyon (1994). ENS-Lyon/LIP N^o 94-21
18. Gao, G.R., Govindarajan, R., Panangaden, P.: Well-behaved dataflow programs for dsp computation. In: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, ICASSP, pp. 561–564 (1992)
19. Geilen, M.: Synchronous dataflow scenarios. ACM Transactions on Embedded Computing Systems **10**(2), 16:1–16:31 (2011)
20. 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, third edn. Springer (2018)
21. Geilen, M., Basten, T., Theelen, B., Otten, R.: An algebra of pareto points. Fundamenta Informaticae **78**(1), 35–74 (2007)
22. Geilen, M., Falk, J., Haubelt, C., Basten, T., Theelen, B., Stuijk, S.: Performance analysis of weakly-consistent scenario-aware dataflow graphs. Journal of Signal Processing Systems **87**(1), 157–175 (2017)
23. Geilen, M., Stuijk, S.: Worst-case performance analysis of synchronous dataflow scenarios. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, CODES/ISSS, pp. 125–134. ACM, New York, NY, USA (2010)
24. Geuns, S.J., Bekooij, M.J.G., Bijlsma, T., Corporaal, H.: Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems. In: Proceedings of Design, Automation and Test in Europe, DATE, pp. 1–6 (2011)
25. Gheorghita, S.V., Stuijk, S., Basten, T., Corporaal, H.: Automatic scenario detection for improved WCET estimation. In: Proceedings of the Design Automation Conference, DAC, pp. 101–104 (2005)
26. Girault, A., Lee, B., Lee, E.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **18**(6), 742 –760 (1999)

27. Griebl, M., Collard, J.F.: Generation of synchronous code for automatic parallelization of while loops. In: S. Haridi, K. Ali, P. Magnusson (eds.) Proceedings of the International Conference on Parallel Processing, EURO-PAR, pp. 313–326. Springer (1995)
28. Griebl, M., Lengauer, C.: A communication scheme for the distributed execution of loop nests with while loops. International Journal of Parallel Programming **23** (1995)
29. Gu, R., Janneck, J.W., Raulet, M., Bhattacharyya, S.S.: Exploiting statically schedulable regions in dataflow programs. In: Proceedings of the International Conference on Acoustics, Speech and Signal Processing, ICASSP, pp. 565–568 (2009)
30. Ha, S., Oh, H.: Decidable signal processing dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
31. Hartmanns, A., Hermanns, H., Bungert, M.: Flexible support for time and costs in scenario-aware dataflow. In: Proceedings of the International Conference on Embedded Software, EMSOFT, pp. 3:1–3:10 (2016)
32. Haykin, S.: Adaptive Filter Theory. Prentice Hall (1996)
33. 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, third edn. Springer (2018)
34. Kahn, G.: The semantics of a simple language for parallel programming. In: Proceedings of Information Processing (1974)
35. van Kampenhout, R., Stuijk, S., Goossens, K.: Programming and analysing scenario-aware dataflow on a multi-processor platform. In: Proceedings of Design, Automation and Test in Europe, DATE, pp. 876–881 (2017)
36. Katoen, J.P., Wu, H.: Exponentially timed SADF: Compositional semantics, reductions, and analysis. In: Proceedings of the International Conference on Embedded Software, EMSOFT, pp. 1–10 (2014)
37. Katoen, J.P., Wu, H.: Probabilistic model checking for uncertain scenario-aware data flow. Transactions on Design Automation of Electronic Systems **22**(1), 15:1–15:27 (2016)
38. Kee, H., Bhattacharyya, S.S., Wong, I., Rao, Y.: 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, ICASSP, pp. 1510–1513 (2010)
39. 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, pp. 1145–1175. Springer (2013)
40. Kienhuis, B., Rijpkema, E., Deprettere, E.: Compaan: Deriving process networks from Matlab for embedded signal processing architectures. In: Proceedings of the International Workshop on Hardware/Software Codesign, CODES, pp. 13–17 (2000)
41. 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)
42. Lin, Y., Choi, Y., Mahlke, S., Mudge, T., Chakrabarti, C.: A parameterized dataflow language extension for embedded streaming systems. In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS, pp. 10–17 (2008)
43. Mattavelli, M., Janneck, J.W., Raulet, M.: MPEG reconfigurable video coding. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, third edn. Springer (2018)
44. Moreira, O.: Temporal analysis and scheduling of hard real-time radios running on a multiprocessor. Ph.D. thesis, Eindhoven University of Technology (2012)
45. Nadezhkin, D., Nikolov, H., Stefanov, T.: Translating affine nested-loop programs with dynamic loop bounds into polyhedral process networks. In: Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia, ESTIMedia, pp. 21–30 (2010)

46. Nadezhkin, D., Stefanov, T.: Automatic derivation of polyhedral process networks from while-loop affine programs. In: Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia, ESTIMedia, pp. 102–111 (2011)
47. Neuendorffer, S., Lee, E.: Hierarchical reconfiguration of dataflow models. In: Proceedings of the International Conference on Formal Methods and Models for Co-Design, 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 Design, Automation and Test in Europe, DATE, pp. 111–116 (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, DAC, pp. 923–926 (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, RSP, pp. 17–23 (2008)
52. Poplavko, P., Basten, T., van Meerbergen, J.: Execution-time prediction for dynamic streaming applications with task-level parallelism. In: Proceedings of the Euromicro Conference on Digital System Design Architectures, Methods and Tools, DSD, pp. 228–235 (2007)
53. Raman, E., Ottoni, G., Raman, A., Bridges, M.J., August, D.I.: Parallel-stage decoupled software pipelining. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO, pp. 114–123 (2008)
54. Rauchwerger, L., Padua, D.: Parallelizing while loops for multiprocessor systems. In: Proceedings of International Parallel Processing Symposium, IPDPS, pp. 347–356 (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 Workshop on Signal Processing Systems, SIPS, pp. 281–286 (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, SoC, pp. 131–136 (2006)
58. Shlien, S.: Guide to MPEG-1 audio standard. IEEE Transactions on Broadcasting **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: Proceedings of the Design Automation Conference, DAC, pp. 917–922 (2009)
60. Skelin, M., Wognsen, E.R., Olesen, M.C., Hansen, R.R., Larsen, K.G.: Model checking of finite-state machine-based scenario-aware dataflow using timed automata. In: Proceedinsg of the International Symposium on Industrial Embedded Systems, SIES, pp. 1–10 (2015)
61. Stefanov, T., Deprettere, E.: Deriving process networks from weakly dynamic applications in system-level design. In: Proceedings of the International Conference on Hardware/Software Codesign and Systems Synthesis, CODES+ISSS, pp. 90–96 (2003)
62. Stefanov, T., Kienhuis, B., Deprettere, E.: Algorithmic transformation techniques for efficient exploration of alternative application instances. In: Proceedings of the International Symposium on Hardware/Software Codesign, CODES, pp. 7–12 (2002)
63. Stuijk, S., Geilen, M., Basten, T.: SDF^3: SDF For Free. In: Proceeding of the International Conference on Application of Concurrency to System Design, ACSD, pp. 276–278 (2006). URL `http://www.es.ele.tue.nl/sdf3`
64. Stuijk, S., Geilen, M., Basten, T.: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. IEEE Transactions on Computers **57**(10), 1331–1345 (2008)

65. Stuijk, S., Geilen, M., Basten, T.: A predictable multiprocessor design flow for streaming applications with dynamic behaviour. In: Proceedings of the Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD, pp. 548–555 (2010)
66. Stuijk, S., Geilen, M., Theelen, B., Basten, T.: Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS, pp. 404–411 (2011)
67. Theelen, B., Geilen, M., Stuijk, S., Gheorghita, S., Basten, T., Voeten, J., Ghamarian, A.: Scenario-aware dataflow. Tech. Rep. ESR-2008-08, Eindhoven University of Technology (2008)
68. Theelen, B., Geilen, M., Voeten, J.: Performance model checking scenario-aware dataflow. In: U. Fahrenberg, S. Tripakis (eds.) Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS, pp. 43–59. Springer (2011)
69. Theelen, B.D.: A performance analysis tool for scenario-aware streaming applications. In: Proceedings of the International Conference on Quantitative Evaluation of Systems, QEST, pp. 269–270 (2007)
70. 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 International Conference on Formal Methods and Models for Codesign, MEMOCODE, pp. 139–148 (2007)
71. 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 the International Conference on Formal Methods and Models for Co-Design, MEMOCODE, pp. 185–194 (2006)
72. Theelen, B.D., Katoen, J.P., Wu, H.: Model checking of scenario-aware dataflow with CADP. In: Proceedings of Design, Automation and Test in Europe, DATE, pp. 653–658 (2012)
73. Turjan, A., Kienhuis, B., Deprettere, E.: Realizations of the Extended Linearization Model. in Domain-Specific Embedded Multiprocessors (Chapter 9), Marcel Dekker, Inc. (2003)
74. Verdoolaege, S.: Polyhedral process networks. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, pp. 1335–1375. Springer (2013)
75. Verdoolaege, S., Nikolov, H., Stefanov, T.: pn: a tool for improved derivation of process networks. EURASIP Journal on Embedded Systems (2007)
76. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., Bruynooghe, M.: Counting integer points in parametric polytopes using Barvinok's rational functions. Algorithmica (2007)
77. Wiggers, M.: Aperiodic multiprocessor scheduling. Ph.D. thesis, University of Twente (2009)
78. 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)
79. Zhai, J.T., Nikolov, H., Stefanov, T.: Modeling adaptive streaming applications with parameterized polyhedral process networks. In: Proceedings of the Design Automation Conference, DAC, pp. 116–121 (2011)