In E. F. Deprettere, J. Teich, and S. Vassiliadis, editors, *Embedded Processor Design Challenges*, Lecture Notes in Computer Science, Springer, pages 1-17. 2002.

Consistency Analysis of Reconfigurable Dataflow Specifications[1]

Bishnupriya Bhattacharya[1] and Shuvra S. Bhattacharyya[2]

[1] Cadence Design Systems
San Jose CA 95134
bpriya@cadence.com

[2] Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies,
University of Maryland, College Park MD 20742, USA
ssb@eng.umd.edu

Abstract. Parameterized dataflow is a meta-modeling approach for incorporating dynamic reconfiguration capabilities into broad classes of dataflow-based design frameworks for digital signal processing (DSP). Through a novel formalization of dataflow parameterization, and a disciplined approach to specifying parameter reconfiguration, the parameterized dataflow framework provides for automated synthesis of robust and efficient embedded software. Central to these synthesis objectives is the formulation and analysis of *consistency* in parameterized dataflow specifications. Consistency analysis of reconfigurable specifications is particularly challenging due to their inherently dynamic behavior. This paper presents a novel framework, based on a concept of *local synchrony*, for managing consistency when synthesizing implementations from dynamically-reconfigurable, parameterized dataflow graphs.

1. Motivation and related work

Dataflow is an established computational model for simulation and synthesis of software for digital signal processing (DSP) applications. The modern trend toward highly dynamic and reconfigurable DSP system behavior, however, poses an important challenge for dataflow-based DSP modeling techniques, which have traditionally been well-suited primarily for applications with significantly static, high-level structure. *Parameterized dataflow* [1] is a promising new meta-modeling approach that addresses this challenge by systematically incorporating dynamic reconfiguration capabilities into broad classes of dataflow-based design frameworks for digital signal processing (DSP).

Through a novel formalization of dataflow parameterization, and a disciplined approach to specifying parameter reconfiguration, the parameterized dataflow framework provides for automated synthesis of robust and efficient embedded software.

1. This research was sponsored by the U. S. National Science Foundation under Grant #9734275.

Central to these synthesis objectives is the formulation and analysis of consistency in parameterized dataflow specifications. Consistency analysis of reconfigurable specifications is particularly challenging due to their inherently dynamic behavior. This paper presents a novel framework, based on a concept of *local synchrony*, for managing consistency when synthesizing implementations from dynamically-reconfigurable, parameterized dataflow graphs. Specifically, we examine consistency issues in the context of dataflow graphs that are based on the *parameterized synchronous dataflow* [1] (PSDF) model of computation (MoC), which is the MoC that results when the parameterized dataflow meta-modeling approach is integrated with the well-known synchronous dataflow MoC. We focus on PSDF in this paper for clarity and uniformity; however, the consistency analysis techniques described in this paper can be adapted to the integration of parameterized dataflow with any dataflow MoC that has a well-defined concept of a graph *iteration* (e.g., to the *parameterized cyclo-static dataflow* model that is described in [2]).

The organization of this paper is as follows. In the remainder of this section, we review a variety of dataflow modeling approaches for DSP. In Section 2, we present an application example to motivate the PSDF MoC, and in Section 3, we review the fundamental semantics of PSDF. In Sections 4 through 7 we develop and illustrate consistency analysis formulations for PSDF specifications, and relate these formulations precisely to constraints for robust execution of dynamically-reconfigurable applications that are modeled in PSDF. In Section 8, we summarize, and mention promising directions for further study.

A restricted version of dataflow, termed *synchronous dataflow* (*SDF*) [12], that offers strong compile-time predictability properties, but has limited expressive power, has been studied extensively in the DSP context. The key restriction in SDF is that the number of data values (*tokens*) produced and consumed by each actor (dataflow graph node) is fixed and known at compile time. Many extensions to SDF have been proposed to increase its expressive power, while maintaining its compile-time predictability properties as much as possible. The primary benefits offered by SDF are static scheduling, and optimization opportunities, leading to a high degree of compile-time predictability. Although an important class of useful DSP applications can be modeled efficiently in SDF, its expressive power is limited to static applications. Thus, many extensions to the SDF model have been proposed, where the objective is to accommodate a broader range of applications, while maintaining a significant part of the compile-time predictability of SDF.

Cyclo-static dataflow (CSDF) and scalable synchronous dataflow (SSDF) are the two most popular extensions of SDF in use today. In CSDF, token production and consumption can vary between actor invocations as long as the variation forms a certain type of periodic pattern [4]. Each time an actor is fired, a different piece of code called a *phase* is executed. For example, consider a *distributor* actor, which routes data received from a single input to each of two outputs in alternation. In SDF, this actor consumes two tokens and produces one token on each of its two outputs. In CSDF, by contrast, the actor consumes one token on its input, and produces tokens according to the periodic pattern 1, 0, 1, 0, … (one token produced on the first invocation, none on the second, and so on) on one output edge, and according to the complementary peri-

odic pattern 0, 1, 0, 1, … on the other output edge. A general CSDF graph can be compiled as a cyclic pattern of pure SDF graphs, and static periodic schedules can be constructed in this manner. CSDF offers several benefits over SDF including increased flexibility in compactly and efficiently representing interaction between actors, decreased buffer memory requirements for some applications, and increased opportunities for behavioral optimizations such as constant propagation and dead code elimination [3, 4].

In SSDF, each actor has the capacity to process any integer multiple of the basic SDF token production (consumption) quantities at an output (input) port, leading to reduced inter-actor context-switching, and hence improved performance in synthesized implementations [16].

In the boolean dataflow (BDF) model, the number of tokens produced or consumed on an edge is either fixed, or is a two-valued function of a control token present on a control terminal of the same actor [6]. Scheduling analysis of a BDF graph can lead to the construction of a *complete cycle*, which is a sequence of actor executions that returns the graph to its original state. Scheduling techniques for BDF graphs attempt to derive *quasi-static schedules* (schedules that are derived using compile time analysis that significantly reduces the amount of run-time scheduling involved) in which each conditional actor invocation is annotated with the run-time condition under which the invocation should occur.

Synchronous piggybacked dataflow (SPDF) is a recently-proposed extension of SDF that provides support for global states in a disciplined fashion. This development of SPDF addresses the problem of updating local parameters ("local states") of a block with global parameters ("global states") based on synchronous state update (SU) requests. SPDF accommodates this by constructing a global table for global parameters, and piggybacking a pointer to a global table entry (tuple of parameter name, and parameter values) on each data sample. A special piggybacking block (PB) is introduced that models the coupling of data samples and the global table pointers. When an SU request is delivered to an actor it will first update its local parameter with a new value of the global parameter before processing its data samples. SPDF utilizes an efficient code synthesis technique with compile-time analysis, such that the PB's function can be simulated without piggybacking (an expensive copy operation), which allows memory efficient code synthesis.

The VSDF [13] and multirate hierarchical timing pair (MHTP) [7] models are dataflow modeling techniques that are geared towards efficient hardware implementation.

Parameterized dataflow modeling differs from dataflow modeling techniques such as SDF, CSDF, SSDF, BDF, SPDF, VSDF, and MHTP in that it is a *meta-modeling* technique: parameterized dataflow can be applied to any underlying "base" dataflow model that has a well-defined notion of a graph iteration (invocation). The dataflow parameterization and parameter value reconfiguration concepts that underlie parameterized dataflow can be incorporated into any dataflow model that satisfies this requirement to significantly increase its expressive power. For example, a minimal periodic schedule is a suitable and natural notion of an iteration in SDF, SSDF, CSDF, and SPDF. Similarly, in BDF, a complete cycle, when it is exists, can be used to spec-

ify a graph iteration.

Furthermore, in contrast to previous work on dataflow modeling, the parameterized dataflow approach achieves increased expressive power entirely through its comprehensive support for parameter definition and parameter value reconfiguration. Actor parameters have been used for years in block diagram DSP design environments. Conventionally, these parameters are assigned static values that remain unchanged throughout execution. The parameterized dataflow approach takes this as a starting point, and develops a comprehensive framework for dynamically reconfiguring the behavior of dataflow actors, edges, graphs, and subsystems by run-time modification of parameter values. SPDF also allows actor parameters to be reconfigured dynamically. However, SPDF is restricted to reconfiguring only those parameters of an actor that do not affect its dataflow behavior (token production/consumption). Parameterized dataflow does not impose this restriction, which greatly enhances the utility of the modeling approach, but significantly complicates scheduling and dataflow consistency analysis. A key consideration in our detailed development of the PSDF MoC (recall that PSDF is the integration of the parameterized dataflow meta-modeling approach with the synchronous dataflow MoC) is addressing these complications in a robust manner, as we will explain in Sections 4 and 7. Such thorough support for parameterization, as well as the associated management of application dynamics in terms of run-time reconfiguration, is not available in any of the previously-developed dataflow modeling techniques.

In recent years, several modeling techniques have been proposed that enhance expressive power by providing precise semantics for integrating dataflow graphs with finite state machine (FSM) models. These include El Greco [5], which provides facilities for “control models” to dynamically configure specification parameters; *charts (pronounced “starcharts”) with heterochronous dataflow as the concurrency model [9]; the FunState intermediate representation [17]; the DF* framework developed at K. U. Leuven [8]; and the control flow provisions in bounded dynamic dataflow [14]. In contrast, parameterized dataflow does not require any departure from the dataflow framework. This is advantageous for users of DSP design tools who are already accustomed to working purely in the dataflow domain, and for whom integration with FSMs may presently be an experimental concept. With a longer term view, due to the meta-modeling nature of parameterized dataflow, it appears promising to incorporate our parameterization/reconfiguration techniques into the dataflow components of existing FSM/dataflow hybrids. This is a useful direction for further investigation.

The parameterized dataflow modeling approach was introduced in [1], which provides an overview of its modeling semantics, and quasi-static scheduling of parameterized dataflow specifications was explored in [2]. This paper focuses on consistency analysis of parameterized dataflow specifications, and develops techniques that can be integrated with scheduling to provide robust operation of synthesized implementations.

2. Application example

To motivate the PSDF model, Fig. 1(a) shows a speech compression application, which is modeled by a PSDF subsystem *Compress*. A speech instance of length L is

transmitted from the sender side to the receiver side using as few bits as possible, applying *analysis-synthesis* techniques [10]. In the init graph, the *genHdr* actor generates a stream of header packets, where each header contains information about a particular speech instance, including its length L. The *setSpch* actor reads a header packet and accordingly configures L, which is modeled as a parameter of the *Compress* subsystem. The *s1* and *s2* actors are "black boxes" responsible for generating samples of this speech instance. In the body graph, actor *s2* generates the speech sample, zero-padding it to a length R. The *An* (*Analyze*) actor accepts small speech segments of size N, and performs linear prediction, producing M auto-regressive (AR) coefficients and the residual error signal of length N at its output. The model order (*ord*) and input length (*len*) parameters of the *An* actor are configured with the subsystem parameters M and N, respectively. The AR coefficients and the residual signal are quantized, encoded (by actors *q1*, *q2*), and transmitted to the receiver side, where these are first dequantized (by actors *d1*, *d2*) and then each segment is reconstructed in the *Sn* (*Synthesize*) actor through AR modeling using the M AR coefficients and the residual signal of length N as excitation. Finally, the *Pl* (*Play*) actor plays the entire reconstructed

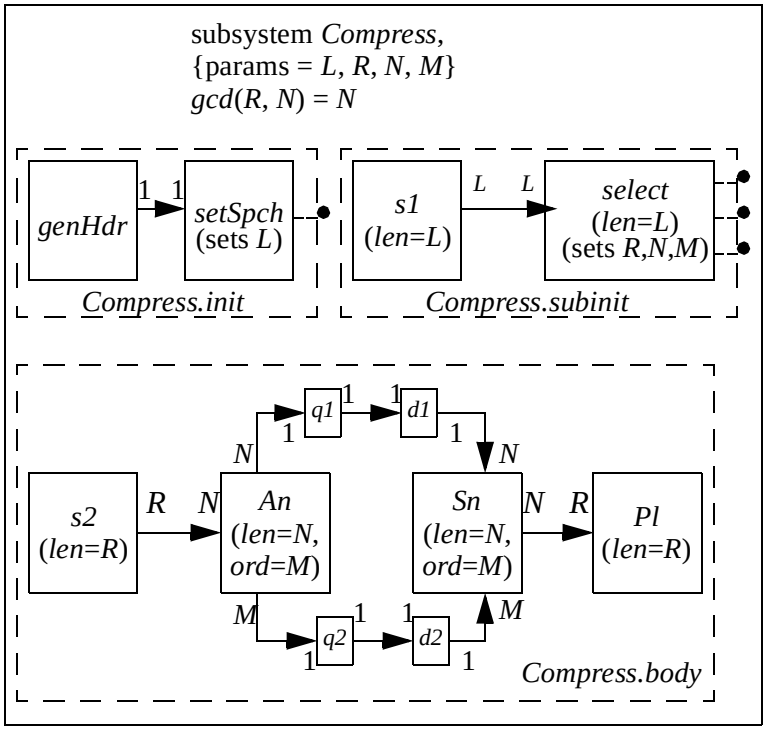

Fig. 1. A PSDF specification of a speech compression application.

speech instance.

The size of each speech segment (N), and the AR model order (M) are important design parameters for producing a good AR model, which is necessary for achieving high compression ratios. The values of N and M, along with the zero-padded speech sample length R are modeled as subsystem parameters of *Compress* that are configured in the subinit graph. The *select* actor in the subinit graph reads the original speech instance, and examines it to determine N and M, using any of the existing techniques, e.g., the Burg segment size selection algorithm, and the AIC order selection criterion [10]. The zero-padded speech length R is computed such that it is the smallest integer greater than L that is exactly divided by the segment size, N. From these relationships, it is useful to convey to the scheduler the assertion that $gcd(R, N) = N$.

Note that for clarity, the above PSDF model does not specify all the details of the application. Our purpose here is to provide an overview of the modeling process, using mixed-grain DSP actors, such that PSDF-specific aspects of the model are emphasized — especially those parameters that are relevant from the scheduler's perspective. All actor parameters that do not affect dataflow behavior have been omitted from the specification. For example, the quantizers and dequantizers will have actor parameters controlling their *quantization* levels and *threshold*s. The *select* actor could determine two such sets — one for the residual and one for the coefficients.

An SDF or CSDF representation of this application will have hard numbers (e.g., 150 instead of N) for the dataflow in Fig. 1(a), corresponding to a particular speech sample. Thus, for processing separate speech samples, the design needs to be modified and the static schedule re-computed. SPDF can accommodate those parameter reconfigurations that do not affect an actor's dataflow properties (e.g., the *threshold* parameter of the *quantizer* actors), but not reconfiguration of the *len* parameter of the *Analyze* actor (*An*), since *len* affects the dataflow of *An*. Thus, again separate designs are necessary to process separate speech samples.

3. PSDF semantics

In the PSDF model, a DSP application will typically be represented as a *PSDF subsystem* Φ that is made up of three *PSDF graphs* — the *init* graph Φ_i, the *subinit* graph Φ_s, and the *body* graph Φ_b. A *set of parameters* is provided to control the behavior of the subsystem. In most cases, the subsystem parameters will be directly derived from the parameters of the application algorithm. For example, in a block adaptive filtering system, the step size and the block size emerge as natural subsystem parameters. Intuitively, in a subsystem, the body graph is used to model its dataflow behavior, while the init and subinit graphs are responsible for configuring subsystem parameter values, thus controlling the body graph behavior.

A PSDF graph is a dataflow graph composed of PSDF actors and PSDF edges. A PSDF actor A is characterized by a set of parameters ($params(A)$), which can control both the functional behavior as well as the dataflow behavior (numbers of tokens consumed and produced) at its input and output *ports*. Each parameter p is either assigned a value from an associated set, called $domain(p)$, or is left unspecified (denoted by the symbol $\perp$). These statically-unspecified parameters are assigned values at run-time that can change dynamically, thus dynamically modifying the actor behavior.

$domain(A)$ defines the set of valid parameter value combinations for A. A configuration that does not assign the value $\perp$ to any parameter is called a *complete configuration*, and the set of all possible complete, valid configurations of $params(A)$ is represented as $\overline{domain(A)}$. Similarly, the sets of valid and complete configurations of a PSDF graph G are denoted $domain(G)$ and $\overline{domain(G)}$, respectively.

Like a PSDF actor, a *PSDF edge* e also has an associated parameterization ($params(e)$), and a set of complete and valid configurations ($\overline{domain(e)}$). The delay characteristics on an edge (e.g., the number of units of delay, initial token values, and re-initialization period) can in general depend on its parameter configuration. In particular, the *delay function* $\delta_e : \overline{domain(e)} \rightarrow \aleph$ associated with edge e gives the delay on e that results from any valid parameter setting.

Now suppose that we have a PSDF graph G, and a complete configuration $C \in \overline{domain(G)}$. Then for a PSDF actor A in G, we represent the instance of A associated with C by $config_{A, C}$, and similarly, for a PSDF edge e, we define $config_{e, C}$ to be the instance of e associated with the complete configuration C. The instance of G associated with C is a pure SDF graph, which we denote by $instance_G(C)$. If the $instance_G(C)$ is sample-rate consistent, then it is possible to compute the corresponding *parameterized repetitions vector* $\mathbf{q}_{G, C}$, which gives the number of times that each actor should be invoked in each iteration of a minimal periodic schedule for $instance_G(C)$.

The *port consumption function* associated with A, denoted $\kappa_A : (in(A) \times \overline{domain(A)}) \rightarrow Z^+$, gives the number of tokens consumed from a specified input port on each invocation of actor A, corresponding to a valid, complete configuration of A. The *port production function* $\varphi_A : (out(A) \times \overline{domain(A)}) \rightarrow Z^+$ associated with A is defined in a similar fashion.

To facilitate bounded memory implementation, the designer must provide a *maximum token transfer function* associated with each PSDF actor A, denoted $\tau_A \in Z^+$, that specifies an upper bound on the number of tokens transferred (produced or consumed) at each port of actor A (per invocation). In contrast to the use of similar bounds in *bounded dynamic dataflow* [14], maximum token transfer bounds are employed in PSDF to *guarantee* bounded memory operation. Similarly, a *maximum delay value*, denoted $\mu_e \in \aleph$, must be specified for a PSDF edge e, which provides an upper bound on the number of delay tokens that can reside at any time on e. The maximum token transfer and delay values are necessary to ensure bounded memory executions of consistent PSDF specifications.

A PSDF subsystem Φ can be embedded within a "parent" PSDF graph abstracted as a *hierarchical PSDF actor* H, and we say that $H = subsystem(\Phi)$. In such a scenario, Φ can participate in dataflow communication with parent graph actors at its *interface ports*. The init graph Φ_i does not participate in this dataflow; the subinit graph Φ_s may only accept dataflow inputs; while the body graph Φ_b both accepts dataflow inputs and produces dataflow outputs. The PSDF operational semantics [1] specify that Φ_i is invoked once at the beginning of each (minimal periodic) invocation of the hierarchical parent graph in which Φ is embedded; Φ_s is invoked at the beginning of each invocation of Φ; and Φ_b is invoked after each invocation of Φ_s.

Consistency issues in PSDF are based on disciplined dynamic scheduling principles that allow every PSDF graph to assume the configuration of an SDF graph on each graph invocation. This ensures that a schedule for a PSDF graph can be constructed as a dynamically reconfigurable SDF schedule. Such scheduling leads to a set of *local synchrony* constraints for PSDF graphs and PSDF subsystems that need to be satisfied for consistent specifications. This paper is concerned with the detailed development of local synchrony concepts for PSDF system analysis, simulation, and synthesis.

A detailed discussion of PSDF modeling semantics can be found in [1], which also shows that the hierarchical, parameterized representation of PSDF supports increased design modularity (e.g., by naturally consolidating distinct actors, in some cases, into different configurations of the same actor), and thus, leads to increased design reuse in block diagram DSP design environments.

4. Local synchrony consistency in PSDF

Consistency in PSDF specifications requires that certain dataflow properties remain fixed across certain types of parameter reconfigurations. This is captured by the following concepts of configuration projections and function invariance.

Definition 1: Given a configuration C of a non-empty parameter set P, and a non-empty subset of parameters $P' \subseteq P$, the *projection of C onto P'*, denoted $C|P'$, is defined by

$$C|P' = \{(p, C(p)) | (p \in P')\}. \quad (1)$$

Thus, the projection is obtained by "discarding" from C all values associated with parameters outside of P'.

Definition 2: Given a parameter set P, a function $f : \overline{domain(P)} \rightarrow R$ into some range set R; and a subset $P' \subseteq P$, we say that f *is invariant over P'* if for every pair $C_1, C_2 \in \overline{domain(P)}$, we have

$$((C_1|(P - P')) = (C_2|(P - P'))) \Rightarrow (f(C_1) = f(C_2)). \quad (2)$$

In other words, f is invariant over P' if the value of f is entirely a function of the parameters outside of P'. Intuitively, the function f does not depend on any member of P', it only depends on the members of $(P - P')$.

The motivation of consistency issues in PSDF stems from the principle of *local SDF scheduling* of PSDF graphs, which is the concept of being able to view every PSDF graph as an SDF graph on each invocation of the graph, after it has been suitably configured. Local SDF scheduling is highly desirable, as it allows a compiler to schedule any PSDF graph (and the subsystems inside it) as a dynamically reconfigurable SDF schedule, thus leveraging the rich library of scheduling and analysis techniques available in SDF. Relevant issues in local SDF scheduling can be classified into three distinct categories — issues that are related to the underlying SDF model, those that relate to bounded memory execution, and issues that arise as a direct consequence of the hierarchical parameterized representation of PSDF. SDF consistency issues such as sample rate mismatch and deadlock detection appear in the first category, while the third category requires that every subsystem embedded in the graph as a hierarchical

actor behave as an SDF actor throughout one invocation of the graph (which may encompass several invocations of the embedded subsystems). Since, in general, a subsystem communicates with its parent graph through its interface ports, the above requirement translates to the necessity of some fixed patterns in the interface dataflow behavior of the subsystem. Since consistency in PSDF implies being able to perform local SDF scheduling, it is referred to as *local synchrony consistency* (or simply local synchrony), and applies to both PSDF graphs and PSDF specifications (subsystems).

More specifically, a PSDF graph G is locally synchronous if for every $p \in \overline{domain(G)}$, the instantiated SDF graph $instance_G(p)$ has the following properties: it is sample rate consistent ($\mathbf{q}_{G,p}$ exists); it is deadlock free; the maximum token transfer bound is satisfied for every port of every actor; the maximum delay value bound is satisfied for every edge; and every child subsystem is locally synchronous.

Formally, this translates to the following *local synchrony* conditions, which must hold for all $p \in \overline{domain(G)}$ in order for the PSDF graph G to be locally synchronous.

- The instantiated SDF graph $instance_G(p)$ has a valid schedule.
- For each actor $v \in V$, and for each input port $\phi \in in(v)$, we have $\kappa_v(\phi, config_{v,p}) \le \tau_v(\phi)$.
- Similarly, for each actor $v \in V$, and for each output port $\phi \in out(v)$, we have $\varphi_v(\phi, config_{v,p}) \le \tau_v(\phi)$.
- For each edge $e \in E$, we have $\delta_e(config_{e,p}) \le \mu_e$.
- For each hierarchical actor H in G, $subsystem(H)$ is locally synchronous.

If these conditions are all satisfied for every $p \in \overline{domain(G)}$, then we say that G is *inherently locally synchronous* (or simply *locally synchronous*). If no $p \in \overline{domain(G)}$ satisfies all of these conditions simultaneously, then G is *inherently locally non-synchronous* (or simply *locally non-synchronous*). If G is neither inherently locally synchronous, nor inherently locally non-synchronous, then G is *partially locally synchronous*. Thus, G is partially locally synchronous if there exists a configuration $p_1 \in \overline{domain(G)}$ for which all of the local synchrony conditions are satisfied, and there also exists a configuration $p_2 \in \overline{domain(G)}$ for which at least one of the conditions is not satisfied. We sometimes separately refer to the different local synchrony conditions as *dataflow consistency* (the existence of a valid schedule), *bounded memory consistency* (the maximum bounds are satisfied for each actor port and each edge), and *subsystem consistency* (each subsystem is locally synchronous) of the PSDF graph G.

Intuitively, a PSDF specification Φ is locally synchronous if its interface dataflow behavior (token production and consumption at interface ports) is determined entirely by the init graph of the specification. As indicated above, local synchrony of a specification is necessary in order to enable local SDF scheduling when the specification is embedded in a graph and communicates with actors in this parent graph through dataflow edges. Four conditions must be satisfied for a specification to be locally synchronous.

First, the init graph must produce exactly one token on each output port on each invocation. This is because each output port is bound to a parameter setting (of the body graph or subinit graph). An alternative is to allow multiple tokens to be produced on an init graph output port, and assign those values one by one to the dependent

parameter on successive invocations of Φ. But this leads to two problems. First, we would have to line up the number of tokens produced with the number of invocations of Φ, thus giving rise to sample rate consistency issues across graph boundaries, which needlessly complicates the semantics. Second, it violates the principle that parameters set in the init graph maintain constant values throughout one invocation of the parent graph of Φ, which in turn violates the requirements for local SDF scheduling. The interface dataflow of the hierarchical actor representing Φ is allowed to depend on parameters set in the init graph. For the parent graph of Φ to be configured as an SDF graph on every invocation, each such embedded hierarchical actor must behave as an SDF actor, for which the parameters set in the init graph must remain constant throughout an invocation of the parent graph.

Similarly the subinit graph must also produce exactly one token on each output port. Parameters set in the subinit graph can change from one invocation of Φ to the next, which is ensured by a single token production at a subinit graph output port on every invocation of the subinit graph. Recall that a single invocation of the subinit graph is followed by exactly one invocation of the body graph. Thus, a token produced on a subinit graph output port is immediately utilized in the corresponding invocation of the body graph. Any excess tokens are redundant (or viewed another way, ambiguous) and will accumulate at the port.

Third, the number of tokens consumed by the subinit graph from each input port must not be a function of the subinit graph parameters that are bound to dataflow inputs of Φ. Finally, the number of tokens produced or consumed at each specification interface port of the body graph must be a function of the body graph parameters that are controlled by the init graph. The third and fourth conditions ensure that a hierarchically nested PSDF specification behaves like an SDF actor throughout any single invocation of the parent graph in which it is embedded, which is necessary for local SDF scheduling.

In mathematical terms, the first condition (called the *init condition* for local synchrony of Φ) is the requirement that

- A. The init graph Φ_i is locally synchronous; and
- B. for each $p \in \overline{domain(\Phi_i)}$, and each interface output port ϕ of Φ_i,

$$\mathbf{q}_{\Phi_i, p}(actor(\phi)) = \varphi_{actor(\phi)}(\phi, config_{actor(\phi), p}) = 1. \quad (3)$$

The init condition dictates that the init graph must be (inherently) locally synchronous and must produce exactly one token at each interface output port on each invocation. Similarly, the second and third conditions are the requirements that

- C. the subinit graph Φ_s is locally synchronous;
- D. for each $p \in \overline{domain(\Phi_s)}$, and each interface output port ϕ of Φ_s,

$$\mathbf{q}_{\Phi_s, p}(actor(\phi)) = \varphi_{actor(\phi)}(\phi, config_{actor(\phi), p}) = 1; \text{ and} \quad (4)$$

- E. for each interface input port ϕ of Φ_s, the product

$$\mathbf{q}_{\Phi_s, p}(actor(\phi))\kappa_{actor(\phi)}(\phi, config_{actor(\phi), p})$$

is invariant over those parameters $p \in params(\Phi_s)$ that are bound to dataflow inputs of Φ.

We refer to Condition D above as the *subinit output condition*, and to Condition E as the *subinit input condition* for local synchrony of Φ. Thus, the subinit graph must be locally synchronous; Φ_s must produce exactly one token at each of its interface output ports on each invocation; and the number of tokens consumed from an input port of Φ_s (during an invocation of Φ) must be a function only of the parameters that are controlled by the init graph or by hierarchically-higher-level graphs.

Finally, the fourth condition for local synchrony of the PSDF specification Φ requires that

- F. the body graph Φ_b is locally synchronous;
- G. for each interface input port ϕ of Φ_b, the product

$$\mathbf{q}_{\Phi_b, p}(actor(\phi))\kappa_{actor(\phi)}(\phi, config_{actor(\phi), p}) \tag{5}$$

is invariant over those parameters $p \in params(\Phi_s)$ that are configured in the subinit graph Φ_s; and

- H. similarly, for each interface output port ϕ of Φ_b, the product

$$\mathbf{q}_{\Phi_b, p}(actor(\phi))\varphi_{actor(\phi)}(\phi, config_{actor(\phi), p}) \tag{6}$$

is also invariant over those parameters $p \in params(\Phi_s)$ that are configured in the subinit graph Φ_s.

Conditions (F), (G) and (H) are collectively termed the *body condition* for local synchrony of Φ. In other words, the body graph must be locally synchronous, and the total number of tokens transferred at any port of Φ_b throughout a given invocation of Φ must depend only on those parameters of Φ_b that are controlled by Φ_i or higher-level graphs.

We sometimes loosely refer to the subinit input condition and the body condition as the local synchrony conditions, and we collectively refer to the requirements of the init condition and the subinit output condition as *unit transfer consistency.*

If Conditions (A) through (H) all hold, then we say that the PSDF specification Φ is *inherently locally synchronous* (or simply *locally synchronous*). If either of the graphs Φ_i, Φ_s, and Φ_b is locally non-synchronous, no $p \in domain(\Phi_i)$ satisfies (3), or no $p \in domain(\Phi_s)$ satisfies (D), then Φ is *inherently locally non-synchronous* (or simply *locally non-synchronous*). If Φ is neither inherently locally synchronous, nor inherently locally non-synchronous, then Φ is *partially locally-synchronous*. Note that if either of the invariance conditions G or H does not hold, then that does not necessarily lead to local non-synchrony of Φ, as the system may satisfy partial local synchrony, which may acceptable if input data sequences that lead to inconsistent parameter reconfigurations do not arise in practice or are very rare.

5. Local synchrony examples

As discussed in Section 4, PSDF subsystems can be classified as *inherently locally synchronous*, *inherently locally non-synchronous*, or *partially locally synchronous*. An illustration of these distinctions is given in Fig. 2. Part (a) shows the body graph of a PSDF specification Φ with one interface input port, and one interface output port. Note that each of the PSDF graphs shown in the figure has two edges and

three nodes. The interface edges (connecting actors in the body graph or subinit graph of a subsystem to parent graph actors) do not contribute to the graph topology in the child (body or subinit) graph. In Fig. 2(a), the body graph parameters p_1 and p_2 are configured in the associated init and subinit graph, respectively. As shown in the figure, the *topology matrix* of Φ_b is a function of the body graph parameters p_1 and p_2. The topology matrix of an SDF graph is a matrix whose rows are indexed by the graph edges, whose columns are indexed by the graph actors, and whose entries give the numbers of tokens produced by actors onto incident output edges, and the negatives of the numbers of tokens consumed by actors from incident input edges (full details on the topology matrix formulation can be found in [12]). Our illustration in Figure 2 extends this concept of the topology matrix to PSDF graphs.

From the repetitions vector $\mathbf{q}$ of Φ_b, the token consumption at the interface input port of the body graph is obtained as $2\,\mathbf{q}(A) = 2p_1$. Similarly, the token production at the interface output port of Φ_b is $\mathbf{q}(C) = 1$. Thus, the interface dataflow of Φ_b is independent of the body graph parameter p_2 that is not configured in Φ_i (i.e., whose value is not updated by the init graph). Hence, the body condition for local synchrony of Φ is satisfied, and if the other local synchrony requirements are also satisfied, then Φ qualifies as an inherently locally synchronous specification.

Fig. 2(b) shows a slightly modified dataflow pattern for Φ_b, such that the token consumption at the interface input port of Φ_b is obtained as $2p_2$, and thus, depends on the parameter p_2, which is configured in the subinit graph. Consequently, Φ is not inherently locally synchronous, rather, it exhibits partial local synchrony with respect

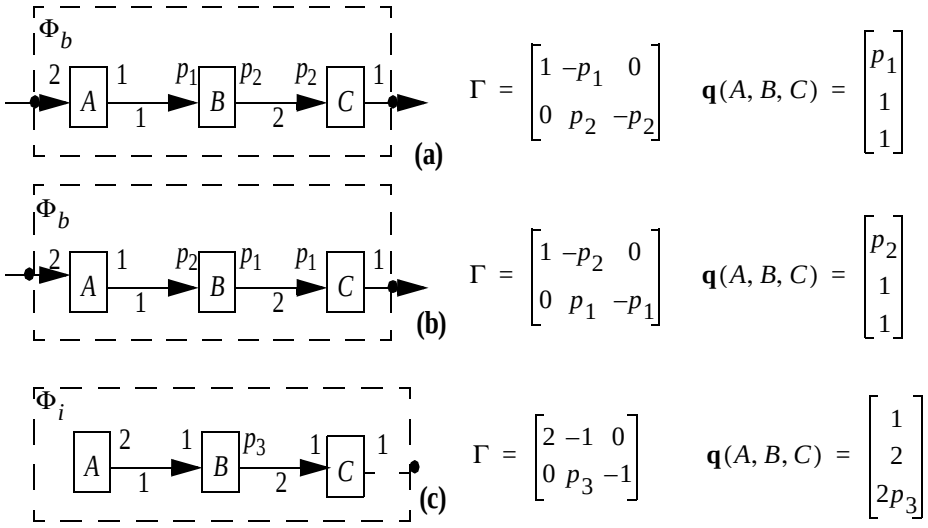

Fig. 2. The symbolic topology matrices and repetitions vectors of three PSDF graphs, used to demonstrate inherent local synchrony, partial local synchrony, and inherent local non-synchrony, respectively. Each dataflow edge is labelled with a positive integer.

to the body condition. If p_2 consistently takes on one particular value at run-time, then a local synchrony error is not encountered. However, if p_2 takes on different values at run-time, then a local synchrony violation is detected, and execution is terminated.

Fig. 2(c) shows the init graph of a specification Φ, which configures a (body or subinit graph) parameter at the interface output port of actor C. From the repetitions vector $\mathbf{x}$ of Φ_i, the number of tokens produced at this interface output port is obtained as $\mathbf{x}(C) = 2p_3$, where p_3 is a parameter of the init graph. Suppose that in this specification, $domain(p_3) = \{1, 2, \ldots, 10\}$. Then whatever value p_3 takes on at run-time, it is clear that Φ_i will produce more than one token at its interface output port on each invocation. Hence, no $C \in \overline{domain(\Phi_i)}$ satisfies the init condition for local synchrony of Φ, and thus, Φ is classified as an inherently locally non-synchronous specification.

6. Binary consistency and decidable dataflow

Before further discussion of analysis and verification issues for PSDF, we first discuss some pre-requisite consistency notions, adapted from [3], for general DSP dataflow specifications.

In general DSP dataflow specifications, the term consistency refers to two essential requirements — the absence of deadlock and unbounded data accumulation. An *inherently consistent* dataflow specification is one that can be implemented without any chance of buffer underflow (deadlock) or unbounded data accumulation (regardless of the input sequences that are applied to the system). If there exist one or more sets of input sequences for which deadlock and unbounded buffering are avoided, and there also exist one or more sets for which deadlock or unbounded buffering results, a specification is termed *partially consistent*. A dataflow specification that is neither consistent nor partially consistent is called *inherently inconsistent* (or simply *inconsistent*). More elaborate forms of consistency based on a probabilistic interpretation of token flow are explored in [11].

A dataflow model of computation is a *decidable* dataflow model if it can be determined in finite time whether or not an arbitrary specification in the model is consistent, and it is a *binary-consistency* model if every specification in the model is either inherently consistent or inherently inconsistent. In other words, a model is a binary-consistency model if it contains no partially consistent specifications. All of the decidable dataflow models that are used in practice today — including SDF, CSDF, and SSDF — are binary-consistency models.

Binary consistency is convenient from a verification point of view since consistency becomes an inherent property of a specification: whether or not deadlock or unbounded data accumulation arises is not dependent on the input sequences that are applied. Of course, such convenience comes at the expense of restricted applicability. A binary-consistency model cannot be used to specify all applications.

7. Robust execution of PSDF specifications

In PSDF, consistency considerations go beyond deadlock and buffer overflow. In particular, the concept of consistency in PSDF includes local synchrony issues. As we have seen in Section 4, local synchrony consistency is, in general, dependent on the input sequences that are applied to the given system. Thus, it is clear that PSDF cannot

be classified as a binary-consistency model. Furthermore, consistency verification for PSDF is not a decidable problem. In general, if a PSDF system completes successfully for a certain input sequence, the system may be inherently consistent, or it may be partially consistent. Similarly, if a PSDF system encounters a local synchrony violation for certain input sequences, the system may be inconsistent or partially consistent.

Since all local synchrony conditions have precise mathematical formulations and at the same time can be checked at well-defined points during run-time operation, the PSDF model accommodates, but does not rely on, rigorous, compile-time verification. There exists a well-defined concept of "well-behaved" operation of a PSDF specification, and the boundary between well-behaved and ill-behaved operation is also clearly defined, and can be detected immediately at run-time in a systematic fashion (by checking local synchrony constraints). More specifically, our development of parameterized dataflow provides a consistency framework and operational semantics that leads to precise and general run-time (or simulation time) consistency verification. In particular, an inconsistent system (a specification together with an input set) in PSDF (or any parameterized dataflow augmentation of one of the existing binary consistency models) will eventually be detected as being inconsistent, which is an improvement in the level of predictability over other models that go beyond binary consistency, such as BDF, DDF, BDDF, and CDDF [18]. In these alternative "dynamic" models, there is no clear-cut semantic criterion on which the run-time environment terminates for an ill-behaved system — termination may be triggered when the buffers on an edge are full, but this is an implementation-dependent criterion. Conversely, in PSDF, when the run-time environment forces termination of an ill-behaved system, it is based on a precisely-defined semantic criterion that the system cannot continue to execute in a locally synchronous manner.

In addition, implementation of the PSDF operational semantics can be streamlined by careful compile-time analysis. Indeed, the PSDF model provides a promising framework for productive compile time analysis that warrants further investigation. As one example of such streamlining, an efficient quasi-static scheduling algorithm for PSDF specifications is developed in [2]. The consistency analysis techniques developed in this paper are complementary to such scheduling techniques. In the general quasi-static scheduling framework of parameterized dataflow, it is possible to perform symbolic computation, and obtain a symbolic repetitions vector of a PSDF graph, similar to what is done in BDF and CDDF. Then depending on how much the compiler knows about the properties of the specification through user assertions, some amount of analysis can be performed on local synchrony consistency. As implied by the operational semantics — which strictly enforces local synchrony — consistency issues that cannot be resolved at compile time must be addressed with run-time verification.

Due to the flexible dynamic reconfiguration capabilities of PSDF, the general problem of statically-verifying (verifying at compile-time) PSDF specifications is clearly non-trivial, and deriving effective, compile-time verification techniques appears to be a promising area for further research. In particular, the issue of compile-time local synchrony verification of a PSDF subsystem calls for more investigation, as it arises as an exclusively PSDF-specific consideration that is inherent in the parameterized hierarchical structure that PSDF proposes. On the other hand, dataflow consis-

tency issues (sample rate consistency and the presence of sufficient delays) are a by-product of the underlying SDF model, and have been explored before in a dynamic context in models such as BDF, CDDF, and BDDF. Compile-time local synchrony verification can take two general forms— determining whether or not a PSDF specification is inherently locally synchronous (in which case run-time local synchrony checks can be eliminated completely), and determining whether or not a specification is inherently locally non-synchronous (in which case the system is unambiguously defective).

Bounded memory execution of consistent applications is a necessary requirement for practical implementations. Given a PSDF specification that is inherently or partially locally synchronous, there always exists a constant bound such that over any admissible execution (execution that does not result in a run-time local synchrony violation), the buffer memory requirement is within the bound. This bound does not depend on the input sequences, and is ensured by bounding the maximum token transfer at an actor port, and the maximum delay accumulation on an edge. BDDF also incorporates the concept of upper bounding the maximum token transfer rate at a dynamic port. However, unlike PSDF, even with these bounds, BDDF does not guarantee bounded memory execution, since it does not possess the concept of a local region of well-behaved operation. In PSDF, inherent and partial local synchrony both ensure bounded memory requirements throughout execution of the associated PSDF system as a sequence of consistent SDF executions. The bound on the token transfer at each actor port ensures that every invocation of a PSDF graph executes in bounded memory, while the bound on the maximum delay tokens on every edge rules out unbounded token accumulation on an edge across invocations of a PSDF graph. A suitable bound on the buffer memory requirements for a PSDF graph $G = (V, E)$ can be expressed as

$$\max_{p \in \nu(G)} \left(\sum_{\theta \in OUT(G)} [(\mathbf{q}_{G,p}(actor(\theta)) \varphi_{actor(\theta)}(\theta, config_{actor(\theta),p}))] + \sum_{e \in G} \delta_e(config_{e,p}) \right), \quad (7)$$

where $OUT(G)$ is the set of actor output ports in G, and

$$\nu(G) = \{p \in \overline{domain(G)} \,|\, \mathbf{q}_{G,p} \text{ exists}\}$$

is simply the set of complete configurations for which G has a valid schedule. From the definition of the maximum token transfer and maximum delay quantities (τ and μ), the quantity in (7) can easily be shown to be less than or equal to the following bound.

$$\sum_{e \in G} \mu_e + \max_{p \in \nu(G)} \left(\sum_{\theta \in OUT(G)} [(\mathbf{q}_{G,p}(actor(\theta)) \tau_{actor(\theta)}(\theta))] \right). \quad (8)$$

The token production and consumption quantities are bounded (by the maximum token transfer function), and the delay on an edge is also bounded (by the maximum delay value), as shown in (8). Since the token transfer at each actor port is bounded, there is only a finite number of possible different values that the repetitions vector can take on. Hence the maxima in (7) and (8) exist. Computing much tighter bounds may in general

be possible, and this appears to be a useful new direction for future work that warrants further investigation.

8. Summary

This paper has developed the concept of local synchrony, and an associated framework for robust execution of reconfigurable dataflow specifications that are based on parameterized dataflow semantics. We have implemented a software tool that accepts a PSDF specification, and generates either a quasi-static or fully-dynamic schedule for it, as appropriate, and in this tool, we have integrated run-time checking of the local synchrony formulations presented here.

Promising directions for future work include modeling and consistency analysis of conditionals (if-then-else constructs) within the PSDF framework; synthesis of streamlined code that implements run-time local synchrony verification; and the development of efficient compile-time algorithms for determining whether or not a PSDF specification is inherently locally synchronous, partially locally synchronous, or inherently defective (locally non-synchronous).

References

1. B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of DSP systems. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1948-1951, June 2000.
2. B. Bhattacharya and S. S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In *Proceedings of the International Workshop on Rapid System Prototyping*, pages 84-89, June 2000.
3. S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for DSP. *IEEE Transactions on Circuits and Systems -- II: Analog and Digital Signal Processing*, 47(9):849-875, September 2000.
4. G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397-408, February 1996.
5. J. T. Buck, and R. Vaidyanathan, "Heterogeneous Modeling and Simulation of Embedded Systems in El Greco," *Proceedings of the International Workshop on Hardware/Software Codesign*, May 2000.
6. J. T. Buck and E. A. Lee. Scheduling dynamic dataflow graphs using the token flow model. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.
7. N. Chandrachoodan, S. S. Bhattacharyya, and K. J. R. Liu. An efficient timing model for hardware implementation of multirate dataflow graphs. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Salt Lake City, Utah, May 2001.
8. N. Cossement, R. Lauwereins, and F. Catthoor. DF*: An extension of synchronous dataflow with data dependency and non-determinism. In *Proceedings of the Forum on Design Languages*, September 2000.
9. A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742-760, June 1999.

10. S. Haykin, *Adaptive Filter Theory*, 3rd edition, Prentice Hall Information and System Sciences Series, 1996.
11. E.A. Lee, "Consistency in Dataflow Graphs," *IEEE Transactions on Parallel and Distributed Systems*, 2(2), April 1991.
12. E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235-1245, September 1987.
13. A. Kerihuel, R. McConnell, and S. Rajopadhye. VSDF: Synchronous data flow for VLSI. Technical Report 843, *Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA)*, 1994.
14. M. Pankert, O. Mauss, S. Ritz, and H. Meyr, "Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1994.
15. C. Park, J. Chung, and S. Ha. Efficient dataflow representation of MPEG-1 audio (layer iii) decoder algorithm with controlled global states. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, 1999.
16. S. Ritz, M. Pankert, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," *Proceedings of the International Conference on Application-Specific Array Processors*, October 1993.
17. L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, "FunState—an internal representation for codesign," *Proceedings of the International Conference on Computer-Aided Design*, November 1999.
18. P. Wauters, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-dynamic dataflow. In *EUROMICRO Workshop on Parallel and Distributed Processing*, January 1996.