Parameterized Dataflow Modeling for DSP Systems

Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya, *Senior Member, IEEE*

***Abstract*—Dataflow has proven to be an attractive computation model for programming digital signal processing (DSP) applications. A restricted version of dataflow, termed *synchronous dataflow* (SDF), that offers strong compile-time predictability properties, but has limited expressive power, has been studied extensively in the DSP context. Many extensions to synchronous dataflow have been proposed to increase its expressivity while maintaining its compile-time predictability properties as much as possible. We propose a parameterized dataflow framework that can be applied as a meta-modeling technique to significantly improve the expressive power of any dataflow model that possesses a well-defined concept of a graph iteration. Indeed, the parameterized dataflow framework is compatible with many of the existing dataflow models for DSP including SDF, *cyclo-static dataflow*, *scalable synchronous dataflow*, and *Boolean dataflow*. In this paper, we develop precise, formal semantics for *parameterized synchronous dataflow* (PSDF)—the application of our parameterized modeling framework to SDF—that allows data-dependent, dynamic DSP systems to be modeled in a natural and intuitive fashion. Through our development of PSDF, we demonstrate that desirable properties of a DSP modeling environment such as dynamic reconfigurability and design reuse emerge as inherent characteristics of our parameterized framework. An example of a speech compression application is used to illustrate the efficacy of the PSDF approach and its amenability to efficient software synthesis techniques. In addition, we illustrate the generality of our parameterized framework by discussing its application to cyclostatic dataflow, which is a popular alternative to the SDF model.**

***Index Terms*—CAD tools, dataflow modeling, embedded systems, reconfigurable design, software synthesis.**

I. Motivation

THE INCREASING use of configurable hardware techniques in digital signal processing (DSP) system design, along with the continual trend toward more dynamic behavior and reconfigurability in DSP applications, is leading to a view of DSP system design as the joint design of *architectures* (design substrates), *parameterizations* (design options that are exposed to higher levels of abstraction), and *configurations* (sets of design choices for the relevant parameterizations) [6], as illustrated in Fig. 1. For example, design of a software-based implementation on a programmable DSP processor can be described as the codesign of the microarchitecture (the architecture), the instruction set (a parameterization of the architecture), and the software that configures the microarchitecture/instruction set pair for the set of application functions. As another example, the design of a digital filtering subroutine can be viewed as the codesign of a software template (e.g., a sequence of for loops), a set of template parameters (e.g., the number of taps, and the filter coefficients), and a set of anticipated parameter value combinations that will frequently be used.

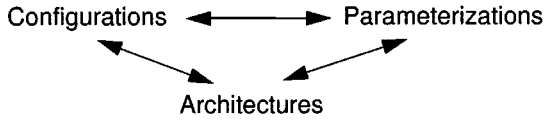


Fig. 1. Abstract view of codesign for DSP.

This trend toward viewing design in terms of configurations of parameterized substrates is rapidly transforming the once-clear separation between hardware and software into a continuum of tradeoffs between specialization and flexibility. It is thus becoming increasingly important to provide precise and powerful mechanisms for modeling parameterization and configurability in design tools for DSP systems. Motivated by this growing need, we develop, in this paper, formal semantics for modeling DSP applications that captures the codesign relationships of Fig. 1 and leads to efficient techniques for automated synthesis of implementations.

In particular, we introduce parameterized dataflow as a meta-modeling technique that can be applied to a wide range of dataflow models to significantly increase their expressive power, such that data-dependent, dynamically reconfigurable DSP systems can be expressed in a natural and intuitive fashion. Our parameterization concepts can be incorporated into any dataflow model in which there is a notion of a graph *iteration*. For example, the parameterized framework is compatible with many of the existing dataflow models such as synchronous dataflow [19], scalable synchronous dataflow [25], cyclo-static dataflow [9], and Boolean dataflow [11]. For clarity and uniformity, and because SDF is currently the most popular and widely studied dataflow model for DSP, we develop parameterized dataflow formally in the context of SDF [called *parameterized synchronous dataflow (PSDF)*].

In addition to the formal model, we have also developed efficient software synthesis techniques (referred to as *quasi-static scheduling*) for a class of PSDF specifications, which is discussed in Section VI-A. A speech compression application modeled in Section VIII shows an example of applying PSDF techniques to real-life DSP designs. In Section IX, we illustrate the generality of the parameterized dataflow framework by dis-

Manuscript received August 9, 2000; revised June 18, 2001. This work was supported in part by the Northrop Grumman Corporation and the National Science Foundation under CAREER award MIP9734275. The associate editor coordinating the review of this paper and approving it for publication was Prof. Chaitali Chakrabarti.

B. Bhattacharya was with the Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 USA. She is now with Cadence Design Systems, San Jose, CA 95134 USA (e-mail: bpriya@cadence.com).

S. S. Bhattacharyya is with the Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 USA (e-mail: ssb@eng.umd.edu).

Publisher Item Identifier S 1053-587X(01)07764-9.

cussing its application to cyclo-static dataflow, which is a popular alternative to the SDF model, in the context of the speech compression application.

A partial summary of a preliminary version of this work has been presented before in [4].

II. Background and Related Work

A. Block-Diagram Design Tools for DSP

Algorithms for DSP are often most naturally described by block diagrams in which computational blocks are interconnected by links that represent sequences of data values. Such block diagram representations have been shown to be highly amenable to the *dataflow* model of computation. In dataflow, a program is described as a directed graph in which vertices (*actors*) represent computations, and edges represent FIFO channels (*buffers*). These channels queue data values (*tokens*) from the output of one actor to the input of another. When an actor is executed (*fired*), it consumes a certain number of tokens from its inputs, and produces a certain number of tokens at its outputs.

A wide variety of commercial DSP design tools have emerged that employ dataflow-based block-diagram programming. These include COSSAP from Synopsys, SPW from Cadence, and ADS from Hewlett Packard. However, due to limitations in synthesis efficiency and expressive power (the range of applications that can be expressed efficiently using the tools), these tools are presently used primarily for simulation and rapid prototyping, and tedious manual fine tuning is still employed to derive final implementations. This paper addresses the issue of significantly increasing expressive power in a manner that is amenable to efficient software synthesis.

B. Notation

In the rest of this paper, we use the following notation. The symbol Z^+ denotes the set of positive integers, and N represents the set of natural numbers {0, 1, 2, ...}. The greatest common divisor of two integers a and b is denoted by $gcd(a, b)$. The notation g: $D \rightarrow R$ represents a function g whose domain and range are D and R, respectively. A directed multigraph G is an ordered pair (V, E), where V is called the *vertex set*, E is called the *edge set*, and associated with each $e \in E$, there are two properties $src(e)$ and $snk(e)$ such that $src(e), snk(e) \in V$.

C. Synchronous Dataflow

Many successful commercial tools for DSP employ synchronous dataflow (SDF) semantics [19] or closely related alternative models, such as scalable synchronous dataflow [25], and cyclo-static dataflow [9]. SDF is a restricted form of dataflow in which the numbers of tokens produced and consumed by each actor execution are restricted to be constant and statically known (known at compile time). This restriction provides SDF important benefits such as predictability, static scheduling, and powerful optimization potential but at the cost of limited expressive power [7].

Fig. 2 shows a simple SDF graph G. Each edge is annotated with the number of tokens produced (consumed) by its source (sink) actor, and the D on the edge from actor A to actor B specifies a unit *delay*. Each unit of delay is implemented as an initial token on the edge. Given an SDF edge e, the source actor, sink actor, and delay of e are denoted by $src(e)$, $snk(e)$, and $d(e)$. In addition, $p(e)$ and $c(e)$ denote the numbers of tokens produced onto e by $src(e)$ and consumed from e by $snk(e)$.

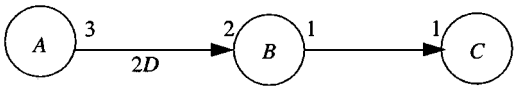


Fig. 2. Simple SDF graph.

In the software synthesis context, many block-diagram programming environments, such as those described in [10] and [18], use the *threading* model [8] to compile an SDF graph. In a threaded approach, the first step is to construct a *valid schedule*—a finite sequence of actor invocations that fires each actor at least once, does not deadlock, and produces no net change in the state of the graph, which is the number of tokens queued on each edge. Corresponding to each actor in the schedule, a code block that is obtained from a library of predefined actors is instantiated. The resulting sequence of code blocks is encapsulated within an infinite loop to generate a software implementation of the SDF graph.

SDF graphs for which valid schedules exist are called *consistent* SDF graphs. Consistent SDF graphs are *sample rate consistent* and *deadlock* free [19]. Lee and Messerschmitt have developed efficient algorithms to determine whether or not a given SDF graph is consistent and to compute the minimum number of times that each actor must be fired in a valid schedule [8]. These minimum numbers of firings are represented by a vector $\mathbf{q}_G$ and indexed by the actors in G (the subscript is suppressed if G is understood). The vector $\mathbf{q}_G$ can be derived by finding the minimum positive integer solution to the *balance equations* for G, which specify that $\mathbf{q}$ must satisfy

$$\mathbf{q}(src(e)) \times p(e) = \mathbf{q}(snk(e)) \times c(e) \quad \text{for every edge } e \text{ in } G. \tag{1}$$

The balance equations can be expressed more compactly in matrix–vector form as $\Gamma\mathbf{q} = 0$, where Γ, which is called the *topology matrix* of G, is a matrix whose rows are indexed by the edges in G, whose columns are indexed by the actors in G, and whose entries are defined by

$$\Gamma(e, A) = \begin{cases} p(e), & \text{if } A = src(e) \\ -c(e), & \text{if } A = snk(e) \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

The vector $\mathbf{q}$, when it exists, is called the *repetitions vector* of G. A schedule S for G is a minimal periodic schedule if it invokes each actor A exactly $\mathbf{q}_G(A)$ times.

The static properties of SDF offer potential for thorough optimization, and effective optimization techniques have been developed in the contexts of data memory minimization [1], joint minimization of code and data [8], [24], [29], high-throughput block processing [25], multiprocessor scheduling (there have been numerous efforts in this category—for example, see [2], [13], [17], [21], [27]), synchronization optimization [23], as well as a variety of other objectives.

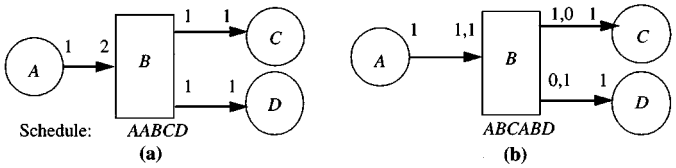


Fig. 3. Cyclo-static dataflow model compared with synchronous dataflow. Actor B is a distributor actor. (a) SDF specification. (b) CSDF specification.

D. Other Dataflow Models

The primary benefits offered by SDF are static scheduling, and optimization opportunities, leading to a high degree of compile-time predictability, as explained in Section II-C. Although an important class of useful DSP applications can be modeled efficiently in SDF [8], [10], [19], 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 firings as long as the variation forms a certain type of periodic pattern [9]. Each time an actor is fired, a different piece of code called a *phase* is executed. Consider the distributor actor shown in Fig. 3 (actor B). This actor 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 two tokens 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 periodic 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 [7].

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 interactor context-switching, and, hence, improved performance in synthesized implementations [25]. The techniques that we develop in this paper for parameterization and dynamic reconfiguration are fully compatible with CSDF and SSDF semantics. We will elaborate on the compatibility with CSDF in Section IX.

In Buck's *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 [11]. It is possible to set up the balance equations for a BDF graph in terms of symbolic variables, and the balance equations can be solved symbolically. This symbolic solution can lead to the detection of a *complete cycle*, which is a sequence of actor executions that returns the graph to its original state. In constructing a schedule for BDF actors, Buck's techniques attempt to derive a quasi-static schedule, where each firing is annotated with the run-time condition under which the firing should occur.

Synchronous piggybacked dataflow (SPDF) was proposed recently by Park *et al.* [22], as an extension of SDF that provides support for *global states*, in a disciplined fashion. Specifically, Park 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 PBs function can be simulated without piggybacking (an expensive copy operation), which allows memory-efficient code synthesis.

Parameterized dataflow modeling differs from dataflow modeling techniques such as SDF, CSDF, SSDF, BDF, and SPDF 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). Our dataflow parameterization concepts can be incorporated into any dataflow model that satisfies this requirement to 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 specify a graph iteration.

Furthermore, in contrast to previous work on dataflow modeling, our 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. Our 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 dynamic reconfiguration of parameter values (see Sections III and IV). SPDF also allows actor parameters to be reconfigured dynamically. However, SPDF *is restricted to reconfiguring only those actor parameters that do not affect its dataflow behavior* (token production/consumption). PSDF does not impose this restriction, which greatly enhances the utility of the model, but significantly complicates scheduling and dataflow consistency analysis. A key consideration in the design of PSDF is addressing these complications in a robust manner, as we will explain in Sections V and VI. 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 pre-

cise semantics for integrating dataflow graphs with finite state machine (FSM) models. These include *El Greco* [12], which provides facilities for "control models" to dynamically configure specification parameters; **charts* (pronounced "starcharts") with *heterochronous dataflow* as the concurrency model [15]; the *FunState* intermediate representation [26]; the *DF** framework developed at K. U. Leuven [14]; and the control flow provisions in *bounded dynamic dataflow* [20]. 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.

III. Parameterized Dataflow Modeling and PSDF

Our parameterized dataflow modeling framework imposes a hierarchy discipline on an underlying dataflow model and allows subsystem behavior to be controlled by sets of parameters that can be configured dynamically. Among the existing dataflow models, SDF has emerged as the most stable and mature model for representing DSP systems. Consequently, we have developed parameterized dataflow formally in the context of SDF, which has resulted in the *parameterized synchronous dataflow* (PSDF) model. PSDF can be viewed as an augmentation of the SDF model that comprehensively incorporates parameterization and run-time management of parameter configurations. In this section, we present an overview of the formal semantics of the PSDF model. Complete details on the concepts introduced here can be found in [3].

A. PSDF Graphs

A PSDF graph is composed of PSDF actors and PSDF edges. A PSDF actor A has a finite set of *input ports* $in(A)$ and a finite set of *output ports* $out(A)$. A PSDF actor is characterized by a set of *parameters* $(params(A))$ that can control the actor's functionality, as well as the actor's dataflow behavior (number of tokens consumed and produced). An application designer determines a *configuration* of a PSDF actor (denoted $config_A$) by assigning values to the parameters of A. 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 $DOMAIN(A)$.

For example, a PSDF *downsampler* actor with input port I and output port O can be characterized by two parameters $params(dnSmpl) = \{factor, phase\}$ that represent, respectively, the decimation ratio and the index of the input token that is actually transferred to the output. The functionality of the downsampler actor depends on both parameters, whereas its dataflow behavior (tokens consumed at input port I) depends only on the *factor* parameter. The domains of these two parameters are given by $domain(factor) = \{1, 2, \ldots, M\}$ and $domain(phase) = \{0, 1, \ldots, M-1\}$, where M is some prespecified maximum integer value, which could, for example, be determined by the maximum word length on the host computer. A configuration of $\{(factor, 5), (phase, 0)\}$ assigns the value 5 to $factor$ and 0 to $phase$. The $phase$ is constrained to take on a value less than $factor$, which is reflected in the actor domain

$$domain(dnSmpl) = \{\{factor, x), (phase, y)\} \mid (x \in domain(factor), y \in domain(phase), y < x)\}.$$

Thus, $\{(factor, 5), (phase, 0)\}$ is a valid configuration for the downsampler actor, but $\{(factor, 5), (phase, 6)\}$ is an invalid configuration.

Fig. 4(a) shows a simple PSDF example that will be used throughout Sections III-B–V to explain PSDF concepts. For the $dnSmpl$ actor, both $factor$ and $phase$ have been left unspecified in its configuration. Actor parameters are indicated within parentheses inside the actor. Parameters of the other actors have been omitted for clarity and are assumed to be statically specified. The $rndInt$ actor produces a random number on each invocation, whereas the $rndInt5$ actor produces five random numbers each time it is invoked. *Propagate* copies its input token to its output, and *print* displays the input it receives.

The *port consumption function* associated with A, which is denoted κ_A: $(in(A) \times DOMAIN(A)) \rightarrow Z^+$ 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. For example, $\kappa_{dnSmpl}(I, \{(factor, 5), (phase, 0)\}) = 5$. The *port production function* φ_A: $(out(A) \times DOMAIN(A)) \rightarrow Z^+$ associated with A is defined in a similar fashion. In general, a software subroutine called the *parameter interpretation function of A*, f_A, that implements κ_A and φ_A, is provided.

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

In order to facilitate bounded memory verification and efficient implementation, the designer must provide a *maximum token transfer function* associated with each PSDF actor A, which is denoted $\tau_A \in Z^+$, that specifies an upper bound on the maximum 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 [20], maximum token transfer bounds are employed in PSDF to

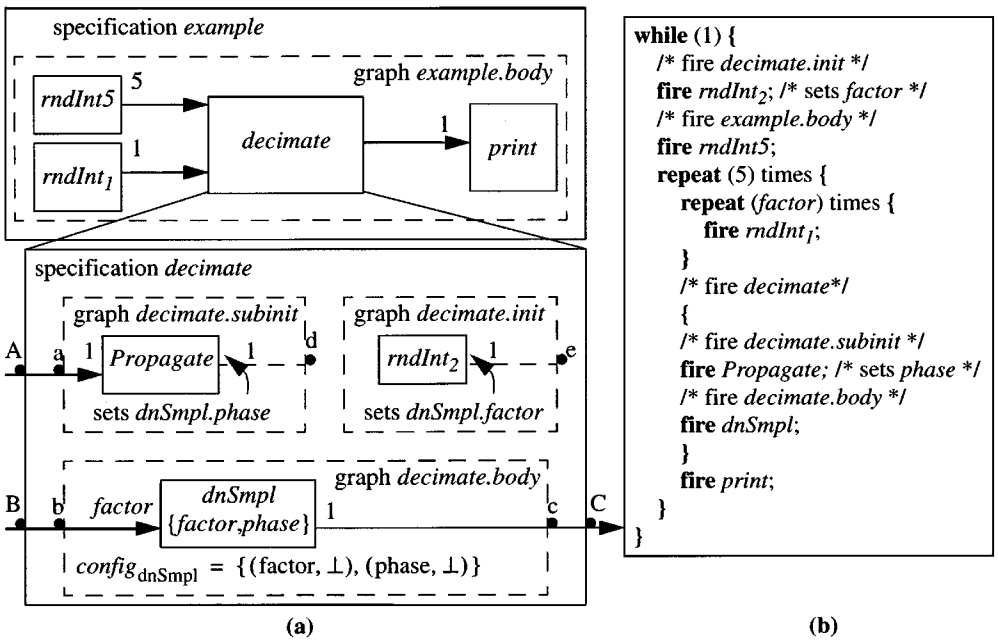


Fig. 4. (a) Example PSDF specification that decimates by a different factor in each run. A PSDF graph (specification) is enclosed in a dashed (solid) rectangle. Interface ports are indicated by bold dots and labeled by letters. In the absence of dataflow, a dashed line connects an actor port to the corresponding graph interface port. (b) Quasi-static schedule for (a). Parameter names have been used without the qualifying graph/specification context for brevity.

guarantee bounded memory operation (through run-time monitoring and verification). Similarly, a *maximum delay value*, which is denoted $\mu_e \in N$, must be specified for a PSDF edge e, and this provides an upper bound on the maximum 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 execution of consistent PSDF specifications. Further details on the use of these maximum values are discussed in Section V and in [3].

A PSDF graph G is an ordered pair (V, E), where each edge $e \in E$ connects an actor output port to an actor input port. The set of input ports in G is given by $IN(V) \equiv \{in(v) | (v \in V)\}$, and similarly, the set of output ports is denoted by $OUT(V) \equiv \{out(v) | (v \in V)\}$. The *internally connected* input and output ports of G (represented by those actor ports on which edges in G are incident) can be proper subsets of $IN(V)$ and $OUT(V)$. In this case, the actor ports on which no edges in G are incident are called the *interface ports* of G. Refer to Fig. 4(a) for examples of graph interface ports (ports a, b, c, d, and e)—port e is an interface output port of the PSDF graph *decimate.init*.

All statically unspecified actor and edge parameters in G propagate "upwards" as parameters of the PSDF graph G, which are denoted $params(G)$, and the set of valid, complete configurations of $params(G)$ is denoted by $DOMAIN(G)$. In Fig. 4(a), the $dnSmpl$ actor parameters $factor$ and $phase$ both become graph parameters of *decimate.body*. Clearly then, given a configuration $C \in DOMAIN(G)$, a pure SDF graph called $instance_G(C)$ emerges by "applying" the configuration C to the unspecified actor and edge parameters in G. In Fig. 4(a), given the configuration $\{(dnSmpl.factor, 5), (dnSmpl.phase, 0)\}$ for the PSDF graph *decimate.body*, clearly, an SDF graph emerges. For a PSDF actor A in G, we represent the instance of A associated with the complete configuration C of G 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 of G. If this instantiated SDF graph $instance_G(C)$ is sample rate consistent, then it is possible to compute the corresponding *parameterized repetitions vector* $\mathbf{q}_{G,C}$.

B. PSDF Specifications

A DSP application will usually be modeled in PSDF through a *PSDF specification*, which is also called a *PSDF subsystem*. A dominant idea in the architecture of our parameterized dataflow framework is the decomposition of a specification (subsystem) into three distinct graphs. Thus, a PSDF specification Φ consists of three PSDF graphs

1) the *init graph* Φ_i;
2) the *subinit graph* Φ_s;
3) the *body* graph Φ_b.

Intuitively, the body graph models the main functional behavior of the specification, whereas the init and subinit graphs control the behavior of the body graph by appropriately configuring the body graph parameters. In Fig. 4(a), the PSDF specification *decimate* is decomposed into the three PSDF graphs

1) *decimate.init*;
2) *decimate.subinit*;
3) *decimate.body*.

PSDF employs a hierarchical modeling structure by allowing a PSDF specification Φ to be embedded in a "parent" PSDF graph G and abstracted as a hierarchical PSDF actor H in this graph. Here, we say that $\Phi = subsystem(H)$. In Fig. 4(a), the PSDF specification *decimate* is embedded as a hierarchical actor in the PSDF graph *example.body*. The specification Φ can, in general, participate in dataflow communication with actors in

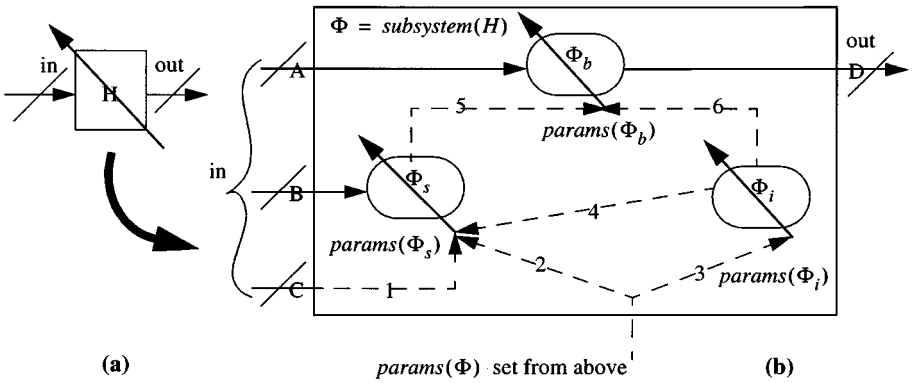


Fig. 5. Operational structure of a PSDF specification. (a) Hierarchical PSDF actor H as it appears externally. (b) Internals of the specification Φ represented by H. A wide tip arrow on a block indicates the existence of parameters of that block that have to be configured externally. A slash on an input or output edge indicates a group of edges. Dataflow is denoted by bold lines, whereas initflow is denoted by dashed lines. Each dataflow path and initflow path is marked, respectively, with a different letter and number.

the parent graph at interface ports of Φ [ports A, B, and C in Fig. 4(a)]. The init graph Φ_i does not take part in this dataflow. The subinit graph Φ_s may only accept dataflow inputs at its interface input ports, whereas the body graph Φ_b may accept dataflow inputs and produce dataflow outputs at its interface ports. The purpose of the init and subinit graphs is to configure parameters, and hence, neither graph produces any dataflow outputs. Instead, the interface output ports of Φ_i and Φ_s are reserved exclusively for configuring parameter values.

The simple example in Fig. 4(a) consists of a topmost specification *example* that is decomposed into a single body graph. The body graph includes a hierarchical actor represented by the specification *decimate*. *decimate* is made up of three graphs

1) init;
2) subinit;
3) body.

The body graph of *decimate* contains the single $dnSmpl$ actor, which accepts external dataflow input at the interface input port B of *decimate* (corresponding to the graph interface input port b of *decimate.body*). Similarly, it also produces dataflow output at interface output port C (graph port c). The subinit graph of decimate accepts external dataflow input at interface input port A (graph port a). The interface output ports of *decimate.init* (port e) and *decimate.subinit* (port d) are used, respectively, to configure the $dnSmpl.factor$ and $dnSmpl.phase$ parameters of *decimate.body*.

Fig. 5 illustrates the operational structure of a PSDF specification Φ embedded as a hierarchical actor H in its parent PSDF graph. As shown in the figure, all the parameters of Φ_b are configured at the interface outputs of Φ_i and Φ_s (paths 6 and 5). Each parameter of Φ_s is configured at an interface output of Φ_i (path 4), is bound to a dataflow interface input port of Φ (path 1, i.e., the value of the input token at this port is assigned as the value of the parameter), or is left unspecified. All the parameters of Φ_i are left unspecified, and along with the unspecified parameters of Φ_s, these parameters propagate "upwards" as the specification parameters of Φ, which are denoted $params(\Phi)$. These specification parameters are configured by the init and subinit graphs of hierarchically higher level subsystems (paths 2 and 3). We refer to this mechanism of parameter configuration as *initflow* to distinguish it from dataflow.

IV. Interaction Between the Init, Subinit, and Body Graphs

Two questions arise naturally from our definition of the PSDF modeling architecture: Why does PSDF provide two separate graphs (init and subinit) to control the body graph behavior, and why is there a difference in the input interface dataflow behavior of these two graphs? The motivation for these conventions is to distinguish between parameter reconfiguration that is allowed to affect subsystem interface dataflow (the number of tokens produced and consumed) and parameter reconfiguration that is restricted to leave interface dataflow unchanged. To maintain a valuable level of predictability and permit efficient quasi-static scheduling, we require that the *interface dataflow of a subsystem must remain unchanged throughout any given iteration of its hierarchical parent subsystem*. Thus, the parent has a consistent view of its module (primitive and hierarchical actors) interfaces throughout any iteration, but the interfaces are allowed to change across iterations of the parent. Parameter reconfiguration that does not change subsystem interface behavior is permitted to occur more frequently—in particular, it is permitted to occur across iterations of the subsystem (rather than the parent subsystem). This gives a subsystem a consistent view of its components' configurations throughout any given iteration.

Thus, in the semantics of PSDF, the subinit graph Φ_s performs reconfiguration activity that, compared to init graph reconfiguration capability, is more restricted in power (what can be changed) but, in general, more frequent. The subinit graph is invoked as an inherent part of the dataflow specification of the parent graph in which Φ is embedded at the beginning of every invocation of Φ; the body graph Φ_b is invoked after each invocation of Φ_s, and the init graph Φ_i is effectively decoupled from the dataflow specification of the parent graph and invoked once, at the beginning of each (minimal periodic) invocation of the parent graph; further details on PSDF activation semantics is given in Section VI. It is thus natural for the subinit graph, but not for the init graph, to accept dataflow inputs from parent graph actors that appear as dataflow predecessors of Φ. In Fig. 4(a), *decimate.subinit* accepts dataflow input from the parent graph actor $rndInt5$ and, accordingly, configures the parameter $dnSmpl.phase$.

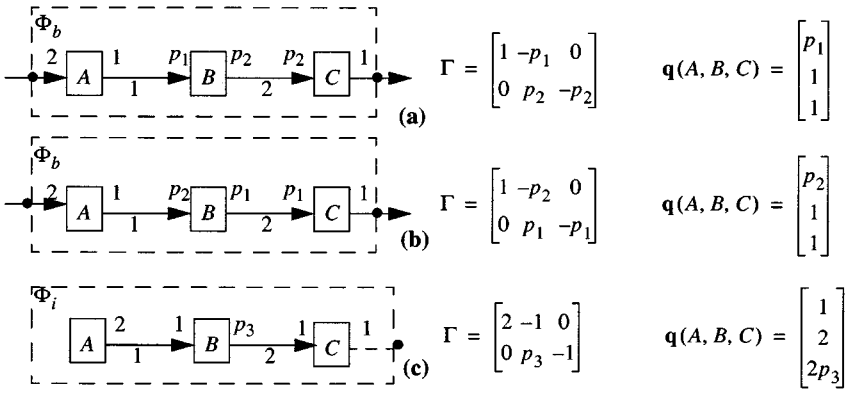


Fig. 6. Symbolic topology matrices and repetitions vectors of three PSDF graphs used to demonstrate inherent local synchrony, partial local synchrony, and inherent local nonsynchrony, respectively. Each dataflow edge is labeled with a positive integer.

In summary, parameter values of Φ_b and Φ_s that are configured by Φ_i (paths 6 and 4 in Fig. 5) remain constant throughout an invocation of the parent graph, whereas parameter values of Φ_b that are configured by Φ_s (path 5 in Fig. 5), and parameter values of Φ_s that are bound to dataflow inputs of Φ (path 1 in Fig. 5) remain constant throughout each invocation of Φ but can change across invocations. Decomposing reconfiguration functionality into separate init and subinit graphs allows the body graph behavior to be controlled at two different levels of granularity, leading to increased flexibility and more expressive power, while maintaining valuable intraiteration predictability. In fact, the PSDF operational semantics allows the designer to configure a parameter with respect to any enclosing subsystem (not just the immediate subsystem or the parent) by appropriate initflow propagation, as we will discuss in Section VII.

For the example in Fig. 4(a), a corresponding quasi-static schedule generated according to the PSDF activation semantics is shown in Fig. 4(b). As seen from the schedule, in each run of the system, a different value of $dnSmpl.factor$ can be assigned in *decimate.init*. Corresponding to a particular value of $dnSmpl.factor$, the *decimate* subsystem is invoked five times, and in each such invocation, $dnSmpl.phase$ is assigned a new value in the subinit graph. In this fashion, corresponding to a particular decimation ratio, the same set of inputs could be processed five times with a different phase, and then, the decimation factor could be changed and the process repeated. Thus, *decimate.subinit* can modify $dnSmpl.phase$ more frequently compared with *decimate.init*'s modification of $dnSmpl.factor$, but $dnSmpl.phase$ cannot control dataflow of decimate, whereas $dnSmpl.factor$ can (and in fact does). Note that in this simple example, we have omitted many details to focus on the main points, e.g., the $rndInt$ actor could have an $upperRange$ parameter, and *example.init* could set up the upper ranges of $rndInt_1$ and $rndInt_2$ appropriately such that $dnSmpl.phase$ is constrained to take on a value less than $dnSmpl.factor$.

V. Local Synchrony

To address consistency analysis of PSDF specifications, we introduce the concept of *local SDF scheduling*, which involves scheduling a PSDF graph as a sequence of pure SDF graphs (corresponding to the sequence of complete configurations that is applied at run time). Local SDF scheduling is highly desirable, as it allows a compiler to schedule any PSDF graph (and the subsystems within it) as a dynamically reconfigurable SDF schedule, thus leveraging the rich library of scheduling and analysis techniques available for SDF. Consistency in PSDF implies the feasibility of local SDF scheduling, and consequently, we refer to it as *local synchrony consistency* (or simply local synchrony). For consistency, both PSDF graphs and PSDF specifications need to satisfy some local synchrony constraints.

The *local synchrony condition* for a PSDF graph $G = (V, E)$ is satisfied for a given $C \in DOMAIN(G)$ if the following conditions all hold.

1) The graph $instance_G(C)$ has a valid schedule, i.e., it is sample rate consistent and deadlock free.
2) For each actor $v \in V$, a) for each input port $\phi \in in(v)$, $\kappa_v(\phi, config_{v,C}) \leq \tau_v(\phi)$, and b) for each output port $\phi \in out(v)$, $\varphi_v(\phi, config_{v,C}) \leq \tau_v(\phi)$, i.e., the maximum token transfer bound is satisfied for every port of every actor.
3) For each edge $e \in E$, $\delta_e(config_{e,C}) \leq \mu_e$, i.e., the maximum delay value bound is satisfied for every edge.
4) For each hierarchical actor H in G, $subsystem(H)$ is locally synchronous; i.e., every child subsystem is locally synchronous.

If the local synchrony condition is satisfied for every $C \in DOMAIN(G)$, we say that G is *inherently locally synchronous* (or simply *locally synchronous*). If no $C \in DOMAIN(G)$ satisfies this requirement, then G is *inherently locally nonsynchronous* (or simply *locally nonsynchronous*). If G is neither locally synchronous nor locally nonsynchronous, then G is *partially locally synchronous*. We sometimes separately refer to the different components of the local synchrony requirement as *dataflow consistency* [property 1)] and bounded memory consistency [properties 2) and 3)] of the PSDF graph G.

Five conditions must be satisfied for a PSDF specification Φ to be locally synchronous. Refer to Fig. 5 for each of these conditions.

1) Each of the PSDF graphs Φ_i, Φ_s, and Φ_b must be locally synchronous.
2) *init condition*: On each invocation, Φ_i must produce **exactly one** token on each interface output port (paths 4 and 6); see Fig. 6 (c) for an example.

```
function execute(graph G, configuration C)
  foreach hierarchical actor H representing subsystem Φ in G
    C_Φinit-set = execute(Φ_i, configure_graph(Φ_i, C))
    C_Φs = configure_graph(Φ_s, merge(C, C_Φinit-set))
    C_Φb = configure_graph(Φ_b, C_Φinit-set)
    precompute_interface_dataflow(Φ_s, C_Φs, Φ_b, C_Φb)
  end for
  foreach leaf actor A in G, apply_configuration(C, config_A)
  foreach edge e in G, apply_configuration(C, config_e)
  compute_repetitions_vector(G); S = compute_schedule(G)
  configuration C_local = ∅; configuration C_out = ∅
  while ((L = get_next_firing(S)) ≠ NULL)
    if (L is a hierarchical actor representing subsystem Φ)
      C_Φsubinit-set = execute(Φ_s, configure_graph(Φ_s, merge(C, merge(C_Φinit-set, C_local))))
      execute(Φ_b, configure_graph(Φ_b, merge(C_Φinit-set, C_Φsubinit-set))
      verify_interface_dataflow(Φ_s, Φ_b)
    else
      execute L
      if (L sets graph parameter p to value v at output port θ)
        if (θ is an interface output port of G)
          update_configuration(C_out, θ, {p, v})
        else
          update_configuration(C_local, θ, {p, v})
        end if
      end if
    end if
  end while
  return C_out
end function
```

Fig. 7. Operational semantics of PSDF.

3) *subinit output condition*: On each invocation, Φ_s must produce **exactly one** token on each interface output port (path 5).
4) *subinit input condition*: The dataflow at each interface input port of Φ_s (path B) must **not** be dependent on those parameters in $params(\Phi_s)$ that are bound to dataflow inputs of Φ (path 1).
5) *body condition*: The dataflow at each interface port (input/output) of Φ_b (paths A and D) must **not** be dependent on those parameters in $params(\Phi_b)$ that are configured in Φ_s (path 5); see Fig. 6(b) for an example.

Conditions 2) and 3) ensure that each interface output port value corresponds to a single new parameter setting, eliminating redundancy or any complication in associating output tokens with parameter values. Conditions 4) and 5) are vitally important in ensuring local SDF scheduling of a PSDF graph. These two conditions guarantee that interface dataflow of Φ is dependent **only** on parameter configurations happening in Φ_i and higher, which implies that once every child init graph in a PSDF graph has been invoked (refer to the operational semantics in Fig. 7), the interface dataflow at every hierarchical actor in G has been determined. Thus, G now consists entirely of SDF actors and can be scheduled as an SDF graph.

For the *decimate* specification in Fig. 4(a), dataflow at the interface input port B depends on the body graph parameter $dnSmpl.factor$, whereas dataflow at any of the interface ports does not depend on the $dnSmpl.phase$ parameter. Thus, for local synchrony, it is necessary for $dnSmpl.factor$ to be configured in *decimate.init* (or higher), whereas it is permissible for $dnSmpl.phase$ to be configured in *decimate.subinit*.

Similar to the corresponding classifications for PSDF graphs, PSDF subsystems can also be classified as *inherently locally synchronous*, *inherently locally nonsynchronous*, or *partially locally synchronous*. An illustration is given in Fig. 6. Fig. 6(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 Fig. 6 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. 6(a), the body graph parameters p_1 and p_2 are configured in the associated init and subinit graph, respectively. As shown in Fig. 6, the topology matrix of Φ_b is a function of the body graph parameters p_1 and p_2. From the repetitions vector 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. 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. 6(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 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. 6(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 of Φ_i, the number of tokens produced at this interface output port is obtained as $\mathbf{q}(C) = 2p_3$, 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 $C \in DOMAIN(\Phi_i)$ satisfies the init condition for local synchrony of Φ, and thus, Φ is classified as an inherently locally nonsynchronous specification.

VI. Operational Semantics

Based on the formalism discussed in Sections III-A–V, a precise operational semantics for PSDF is given in Fig. 7 in a pseudo-code format. Fig. 7 shows the routine `execute`, which, given a PSDF graph G and a complete configuration C for $params(G)$, computes and executes a schedule for the instantiated SDF graph $instance_G(C)$ and verifies its local synchrony. The routine returns the output configuration C_{out} determined for those parameters that are configured at the interface output ports of G. Execution of a top-level PSDF specification Φ is initiated by invoking the recursive routine `execute` on the graph G in which Φ is implicitly assumed to be embedded, with an empty configuration C.

To compute a schedule for G, it is necessary to obtain the interface dataflow of each embedded subsystem Φ that appears as a hierarchical actor H in G. The first step invokes the init graph of Φ, where the init graph's configuration is extracted as a subset of the configuration C of its parent graph G. This returns in $C_{\Phi\text{init-set}}$ a configuration of those body and subinit graph parameters of G that are set by the init graph. C and $C_{\Phi\text{init-set}}$ are then used to compute a configuration for Φ_s and Φ_b. In this process, those parameters of Φ_s and Φ_b that are not configured in Φ_i (through $C_{\Phi\text{init-set}}$) or in ancestor subsystems (through C) have unknown values and are assigned *default values* (as specified statically by the application programmer) in order to determine complete configurations for Φ_s and Φ_b. Using these complete configurations, the routine `precompute_interface_dataflow` configures Φ_s and Φ_b as SDF graphs, computes the repetitions vector of each, and obtains the dataflow (numbers of tokens consumed and produced) at interface ports of Φ_s and Φ_b. G is now configured as an SDF graph by resolving the remaining unknown actor/edge parameter values, following which it is straightforward to compute its schedule S using SDF techniques.

The next step is to fire actors in the order specified by the schedule. Firing a leaf actor L implies executing (the code of) the actor. If L configures a graph parameter at one of its output ports θ, then either the output configuration C_{out} (to be returned after G finishes execution) or the local configuration C_{local} (to be used in configuring the subinit graph of an embedded subsystem) is augmented, depending on whether or not θ is an interface port of G. If L is a hierarchical actor, then the associated subinit graph is executed first, followed by executing the body graph. An execution step is preceded by computing a complete configuration for each graph; note that these configurations of Φ_s and Φ_b are based on *actual* values of all graph parameters, instead of using default values for some graph parameters, as was done while precomputing their interface dataflow.

Among the relevant verification tasks, checking dataflow consistency and bounded memory consistency of G and verifying the init condition and subinit output condition of a subsystem Φ embedded in G are straightforward. Verification of the body condition and subinit input condition for local synchrony of subsystem Φ is done in the routine `verify_interface_dataflow`, which compares the precomputed dataflow obtained at the interface ports of Φ_s and Φ_b (after firing Φ_i) with the actual dataflow obtained after firing Φ_s and Φ_b. Thus, the parameters of Φ_s and Φ_b that are not configured in Φ_i (or in hierarchically higher level subsystems) must have default values specified judiciously, such that their inter-relationships in determining the interface dataflow of Φ are the same as any combination of values that these parameters can take on at run time.

The complexity of function `execute` in Fig. 7 is dominated by the schedule computation step (`compute_schedule`). Using techniques related to the family of *loose interdependence algorithms* [8], schedules usually can be constructed in $O(en)$ time, where e and n are the numbers of edges and actors in the associated PSDF graph. We say "usually" because this holds whenever the corresponding instantiated SDF graph does not contain any subgraphs of a certain form called *tightly interdependent subgraphs* [8]. If tightly interdependent subgraphs are present, they require additional $O(I_T)$ time to schedule, where $I_T = \sum_{A \in T} \mathbf{q}(A)$, and T is the set of actors that are contained in tightly interdependent subgraphs. In practice, however, tightly interdependent components are extremely rare [8].

Quasi-static scheduling approaches can be used to streamline the scheduling phase of the PSDF operational semantics significantly beyond the efficiency achieved by loose interdependence algorithm techniques. We discuss this further in Section VI-A.

One of the most useful qualities of PSDF is the robustness of its operational semantics, which accommodates, but does not depend on, rigorous consistency verification at compile time. There is a precise 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 an efficient fashion. In particular, an inconsistent system (a specification together with an input set) in PSDF (or any parameterized augmentation of one of the existing statically schedulable models) will eventually be detected as being inconsistent, which is a significant improvement in the level of predictability over other models that go beyond static schedulability, such as BDF [11], cyclo-dynamic dataflow [28], and bounded dynamic dataflow [20]. In these alternative "dynamic" models, there is no clear semantic criterion on which execution terminates for an ill-behaved system; termination may be triggered if the buffer on an edge overflows, 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 precise semantic criterion that the system cannot continue to operate in a locally synchronous manner.

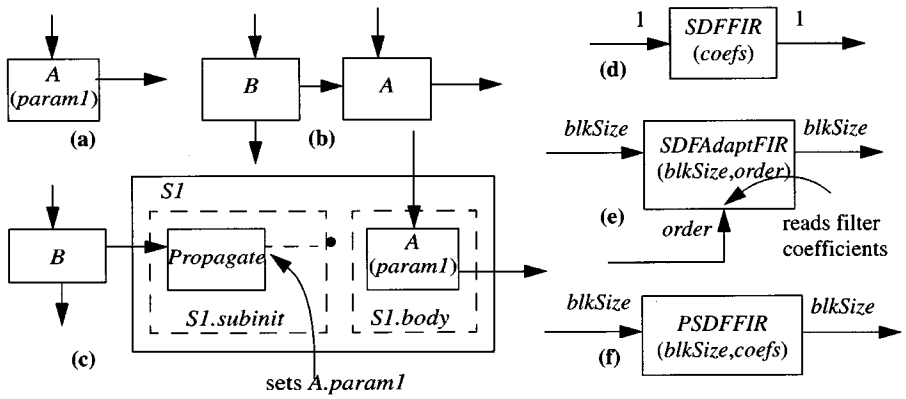


Fig. 8. Example to demonstrate that dataflow inputs can be simulated by actor parameters, leading to condensed actor libraries in block diagram-based DSP design environments.

A. Efficient Implementation: Quasi-static Scheduling

According to the operational semantics, a schedule for a PSDF graph needs to be recomputed for every parameter reconfiguration, which can result in significant overhead. However, implementation of the PSDF operational semantics can be streamlined by careful compile-time analysis. Indeed, the PSDF model and the associated local synchrony concept provide a promising framework for productive compile time analysis that warrants further investigation. As one example of such streamlining, our implementation of the PSDF operational semantics incorporates an efficient *quasi-static* scheduling technique for a class of PSDF specifications. Quasi-static schedules are generated at compile time, and generally, they fix a significant portion of scheduling decisions at compile time but may contain code that performs some data-dependent computations at run time.

The quasi-static scheduling technique can be applied on all acyclic PSDF specifications and on a class of cyclic PSDF specifications that satisfy certain technical constraints in their feedback loops [3], [5]. For such cyclic graphs, the feedback loops can effectively be broken, resulting in acyclic PSDF specifications for the purpose of scheduling. Fortunately, a large class of practical DSP applications fall under the classes of graphs accommodated by our quasi-static scheduling approach [3], [5].

Our scheduling technique is based on a an extension of a *clustering* algorithm developed for SDF graphs called acyclic pairwise grouping of adjacent nodes (APGANs) [8]. Given a PSDF graph, the basic clustering step in parameterized APGAN (P-APGAN) collapses two adjacent actors into a single actor, adjusts the graph topology/dataflow accordingly, and performs symbolic computation (with unspecified parameters) to determine a minimal periodic schedule for the two-actor cluster. This basic clustering step is repeated until the whole graph is reduced to a single actor. The cluster hierarchy is then traversed recursively, and a *parameterized looped schedule* [5] is generated at compile time. In addition, according to the PSDF operational semantics, P-APGAN inserts necessary preamble code to configure parameters and perform local synchrony checks in the generated schedule. Examples of quasi-static schedules obtained using our approach are given in Figs. 4(b) and 9(b).

APGAN produces provably optimal results with respect to joint code size and buffer size minimization for a class of SDF graphs [8]. In going from SDF to PSDF, P-APGAN accommodates greater expressive power and, hence, less compile-time knowledge, thus losing some of the optimization features of the original APGAN. Instead, P-APGAN utilizes heuristics with the objective of minimizing code size and buffer size, as well as reducing run-time overhead in configuring parameters and verifying local synchrony. For further details on this scheduling approach, see [3] and [5]. We have implemented a software tool that accepts a PSDF specification and generates either a quasi-static or a run-time schedule for it, as appropriate.

VII. Role of Parameters

In the dataflow framework, an actor accepts dataflow inputs, which can change (take on different values) across every invocation of the actor and, thus, can control the behavior of the actor at the granularity of every actor invocation. In addition to this form of behavior control, the PSDF model allows an actor's behavior to be controlled by parameters that are configured from an enclosing subsystem. In PSDF, an actor parameter is configured once per iteration of an enclosing subsystem and, thus, maintains a constant value for a sequence of successive invocations of the actor. Hence, parameters generally control actor behavior at a coarser level of granularity than dataflow inputs. However, the semantics of PSDF allow controlling actor parameters through dataflow inputs, thus allowing actor behavior to be controlled at the granularity of every actor invocation.

An example is given in Fig. 8. Fig. 8(a) shows a PSDF actor A with a single parameter $param1$ and one input and output port. Suppose that this specification of A is provided as part of a predefined actor library in some DSP block diagram programming environment, and an application requires that $param1$ change across every invocation of A, based on dataflow input from another actor, say, B. One obvious solution [see Fig. 8(b)] is to change the library specification of A by adding an extra input port, where it accepts dataflow from B, which is then used to replace the functionality performed by $param1$. Fig. 8(c) shows a different solution, utilizing PSDF subsystem semantics, that *does not require a separate version of the original library specification of* A. Here, A is encapsulated inside the body graph of a new subsystem $S1$, and B provides dataflow input to the *Propagate* actor in the subinit graph of $S1$ that configures $param1$ at

its output port. According to PSDF semantics, every invocation of A is preceded by an invocation of *Propagate*, which configures $param1$ with the dataflow output of B, thus effectively allowing $param1$ to be controlled by dataflow input.

As a more concrete example, consider Fig. 8(d) and (f). As illustrated in Fig. 8(d) and (e), block diagram DSP environments, such as the SDF-based domains in Ptolemy [10], typically provide separate SDF models of a simple FIR filter (for processing a single input token, with the filter coefficients represented as a vector-parameter), and an adaptive FIR filter (for processing a block of input data on each run with the filter coefficients obtained dynamically from an additional input port). In contrast, with PSDF, the flexible dynamic parameter reconfiguration capability makes it possible to replace the structures of Fig. 8(d) and (e) with a single FIR filter model [Fig. 8(f)], whose functionality can be appropriately configured.

In fact, a PSDF actor parameter can be controlled at the granularity of any enclosing object (an actor, subsystem, or graph). For example, if a body graph parameter is configured in the associated subinit graph or a subinit graph parameter is bound to a dataflow input of the subsystem, then the parameter changes across every invocation of the associated subsystem, whereas a subinit graph parameter that is set by initflow from a hierarchically higher level subsystem changes across every invocation of that ancestor subsystem. Hence, with a library specification of actor A as in Fig. 8(a), the application designer can fulfill various application-specific needs by configuring $param1$ at the appropriate level of hierarchy, which includes assigning a static value to $param1$ to be maintained over all invocations of A, holding $param1$ constant over a certain number of invocations of A with respect to a single invocation of any enclosing subsystem (e.g., its parent) but allowing it to change across this window, and allowing $param1$ to change across every invocation of A. This translates to increased design flexibility and design reuse and eliminates the need to increase the size of the actor library by adding different "versions" of the same actor, as in Fig. 8(b).

VIII. Application Example: Speech Compression

A. Specifying Parameter Configuration

Before presenting an application example, we would like to clarify the exact mechanism of parameter (re)configuration employed in PSDF. In Sections III–VII, we have presented a bottom-up model of actor, edge, graph, and specification parameters. We realize that from an application designer's perspective, a top-down model—using direct subsystem parameters (possibly derived from parameters of the algorithm itself) and configuring actor/edge parameters with subsystem parameter values—may sometimes be more natural. For a user-friendly front end, we allow this top-down approach in designing applications. An exact mapping between these two approaches is given in [3].

Note that in either case, parameter configuration is not the block designer's job, but rather the application designer's responsibility. The parameter configuration information is factored into the PSDF code synthesis system for that application. In the bottom-up model, the application designer specifies at an init/subinit interface output port which actor/edge parameter(s) that port configures. For example, Fig. 8(c) specifies that the interface output port of the *Propagate* actor in $S1.subinit$ configures $param1$ of actor A in $S1.body$. In the top-down model, parameter configuration is a two-step process. At each init/subinit interface output port, the application designer specifies which subsystem parameter(s) that port configures. In addition, an actor parameter is assigned a subsystem parameter value in order to be dynamically reconfigured. Subsystem parameter values are visible "downwards" in the children of the associated subsystem. Examples of top-down parameter configuration can be seen in Figs. 9(a) and 10(a). For brevity, we have indicated parameter configuration as happening inside the actors (see actors *select* and *An*), but keep in mind that these configurations are actually part of this particular application and not of the actors.

B. Speech Compression

Fig. 9(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 [16]. 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 $d1$ and $d2$), 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 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 [16]. 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

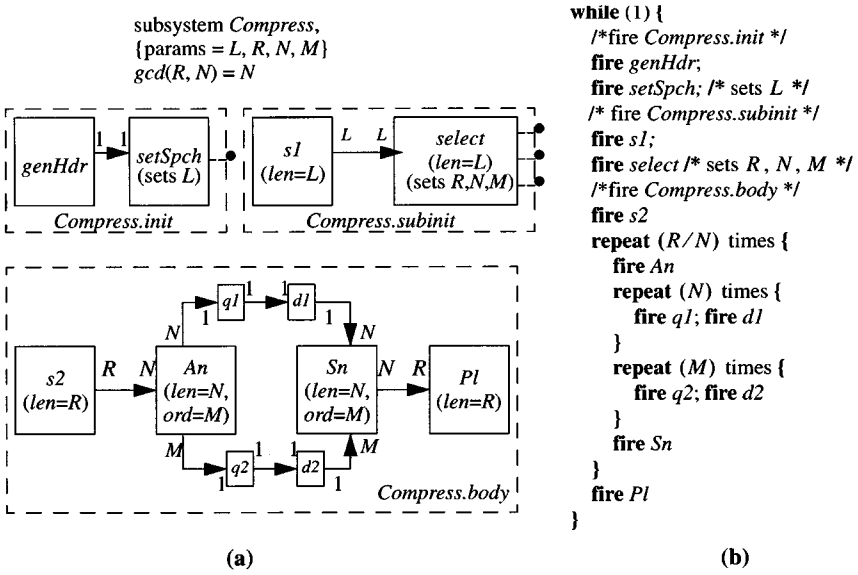


Fig. 9. (a) PSDF specification of a speech compression application. (b) Quasi-static schedule for the specification.

N. This fact is conveyed to the scheduler through the user assertion $gcd(R, N) = N$.

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 *thresholds*. The *select* actor could determine two such sets: one for the residual and one for the coefficients. Further details are available in [3], which also documents an alternative PSDF specification, where a speech instance is generated only once instead of twice, as in this case. More examples of DSP applications modeled in PSDF can be found in [3].

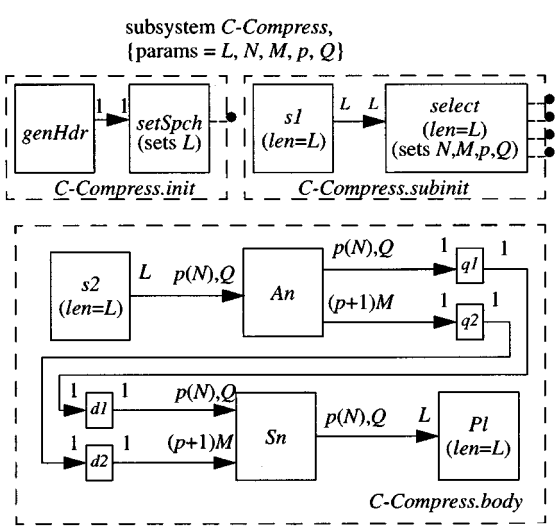
The quasi-static schedule for the *Compress* specification, which is determined by our quasi-static scheduler implementation, is shown in Fig. 9(b). This schedule utilizes the property that N exactly divides R, which can easily be asserted to the compiler by the designer, as discussed previously. The application can be run in an infinite loop, and in each run, a different speech instance can be processed *reusing the same design and without having to recompile the software.*

An SDF or CSDF representation of this application will have hard numbers (e.g., 150 instead of N) for the dataflow in Fig. 9(a), corresponding to a particular speech sample. Thus, for processing separate speech samples, the design needs to be modified and the static schedule recomputed. SPDF can accommodate those actor parameter reconfigurations that do not affect its dataflow (e.g., the *threshold* parameter of the *quantizer* actors) but not reconfiguration of the len parameter of the *Analyze* actor (An) since len affects An's dataflow. Thus, again, separate designs are necessary to process separate speech samples. A fully dynamic model like *dynamic dataflow* (DDF) can model this application, but it cannot generate a quasi-static schedule as in Fig. 9(b). Instead, all actor firings will be determined at run time, incurring considerable overhead. This demonstrates that only PSDF accommodates both increased expressive power along with efficient scheduling, thus achieving a unique balance.

IX. Parameterization as a Meta-Modeling Technique

The PSDF model applies our parameterized dataflow concept to the synchronous dataflow formalism. As discussed in Section III, it is also possible to apply the same parameterization techniques to other dataflow models that have well-defined notions of a graph iteration and obtain similar dynamically-reconfigurable model augmentations. For example, cyclo-static dataflow (CSDF) can be extended to a parameterized cyclo-static dataflow (PCSDF) model that has the same appealing reconfiguration-related properties as PSDF. An illustration is given in Fig. 10 (a) that models the speech compression application of Section VIII-B in PCSDF. Recall that in the PSDF specification, an instance of the speech sample of length L must be zero padded to a length R such that the size of each segment (N) exactly divides the zero-padded length. PSDF inherits this necessity of zero padding from the underlying SDF model, and in the PCSDF specification, this zero padding is no longer necessary. Instead of the zero-padded length R, we have two other parameters: p, which gives the number of segments of size N contained in the original speech sample, and Q, which represents the size of the residual segment. Thus, if L is divided by N, then p represents the quotient, and Q represents the remainder. As before, M gives the model order of the AR model of each speech segment.

In the PCSDF specification *C-Compress*, which is shown in Fig. 10(a), the code for the An and Sn actors has been decomposed into phases so that the dataflow can vary even in the same invocation of the parent graph, unlike the PSDF version, where the dataflow varies only across invocations of the parent graph. The notation $p(N)$, Q denotes the parameterized




```
/* fire C-Compress.body */
fire s2
repeat (p) times {
  fire An
  repeat (N) times {
    fire q1; fire d1
  }
  repeat (M) times {
    fire q2; fire d2
  }
  fire Sn
}
fire An
repeat (Q) times {
  fire q1; fire d1
}
repeat (M) times {
    fire q2; fire d2
  }
fire Sn
fire Pl
```

(b)

Fig. 10. (a) PCSDF specification of the speech compression application. (b) Quasi-static schedule for the body graph of the specification.

cyclo-static dataflow sequence $N, N, \ldots, N, Q$ with N repeated p times. Similarly, $(p+1)M$ denotes a sequence of the form $M, M, \ldots, M$ in which M is repeated $(p+1)$ times. The token consumption pattern $p(N), Q$ signifies that the first p invocations of An consume N tokens each from the input port, and the $(p+1)$th invocation consumes Q tokens.

A possible PCSDF quasi-static schedule for the body graph of *C-Compress* is given in Fig. 10(b). The actor invocations enclosed in the first p firings process the first p segments of the speech sample, each of length N. The next block of actor executions processes the residual segment of length Q.

X. Conclusions

We have introduced a parameterized dataflow framework that can be applied as a meta-modeling technique to significantly increase the expressive power of a wide range of dataflow models, and we have developed in detail the formal semantics of *parameterized synchronous dataflow* (PSDF).

Parameterization of subsystem functionality emerges as a natural concept from the application modeling viewpoint, and combined with the underlying SDF model that has proven to be very well-suited for designing static DSP systems, this makes PSDF a natural and intuitive choice for modeling data-dependent, dynamic DSP systems. The parameterized framework also supports increased design reuse, leading to condensed actor libraries for block-diagram DSP programming environments. PSDF possesses robust and elegant operational semantics, and efficient quasi-static schedules can be constructed for a class of specifications.

A promising direction for future work is modeling conditionals (*if–then–else*) within the PSDF framework of dynamically reconfigurable parameters to further increase its expressive power. Formal verification and optimized software synthesis from PSDF specifications are other interesting areas for future work.

References

[1] M. Ade, R. Lauwereins, and J. A. Peperstraete, "Data memory minimization for synchronous data flow graphs emulated on DSP-FPGA targets," in *Proc. Des. Automat. Conf.* , June 1994.

[2] S. Banerjee, T. Hamada, P. M. Chau, and R. D. Fellman, "Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems," *IEEE Trans. Signal Processing*, vol. 43, pp. 1468–1484, June 1995.

[3] B. Bhattacharya, "Parameterized modeling and scheduling for dataflow graphs," M.S. thesis, Dept. Elect. Comput. Eng., Univ. Maryland, College Park, Dec. 1999.

[4] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling of DSP systems," in *Proc. Int. conf. Acoust., Speech, Signal Process.*, June 2000.

[5] ——, "Quasi-static scheduling of re-configurable dataflow graphs for DSP systems," in *Proc. Int. Workshop Rapid Syst. Prototyping*, June 2000.

[6] S. S. Bhattacharyya, "Hardware/software co-synthesis of DSP systems," in Programmable Digital Signal Processors: Architecture, Programming, and Applications, Y. H. Hu, Ed. New York: Marcel Dekker, to be published.

[7] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for DSP," *IEEE Trans. Circuits Syst. II*, vol. 47, pp. 849–875, Sept. 2000.

[8] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Boston, MA: Kluwer, 1996.

[9] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Trans. Signal Processing*, vol. 44, pp. 397–408, Feb. 1996.

[10] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Comput. Simulation*, Apr. 1994.

[11] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, Apr. 1993.

[12] J. T. Buck and R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in El Greco," in *Proc. Int. Workshop Hardware/Software Codes.*, May 2000.

[13] L. F. Chao and E. Sha, "Unfolding and retiming data-flow DSP programs for RISC multiprocessor scheduling," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, Apr. 1992.

[14] N. Cossement, R. Lauwereins, and F. Catthoor, "DF*: An extension of synchronous dataflow with data dependency and nondeterminism," in *Proc. Forum Des. Languages*, Sept. 2000.

[15] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Trans. Comput.-Aided Des.*, vol. 18, pp. 742–760, June 1999.

[16] S. Haykin, *Adaptive Filter Theory*, 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[17] P. Hoang and J. Rabaey, "A compiler for multiprocessor DSP implementation," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, Mar. 1992.
[18] E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. S. Bhattacharyya, "Gabriel: A design environment for DSP," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 37, pp. 1751–1762, Nov. 1989.
[19] E. A. Lee and D. G. Messerschmitt, "Pipeline interleaved programmable DSP's: Synchronous dataflow programming," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-35, Sept. 1987.
[20] M. Pankert, O. Mauss, S. Ritz, and H. Meyr, "Dynamic dataflow and control flow in high level DSP code synthesis," in *Proc. Int. Conf. Acoust., Speech, Signal Processing*, Apr. 1994.
[21] K. K. Parhi and D. G. Messerschmitt, "Static rate-optimal scheduling of iterative dataflow programs via optimum unfolding," *IEEE Trans. Comput.*, vol. 40, pp. 178–195, Feb. 1991.
[22] C. Park, J. Chung, and S. Ha, "Efficient dataflow representation of MPEG-1 audio (Layer III) decoder algorithm with controlled global states,," in *Proc. IEEE Workshop Signal Process. Syst.: Des. Implement.*, Oct. 1999.
[23] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. New York: Marcel Dekker, 2000.
[24] W. Sung, J. Kim, and S. Ha, "Memory efficient synthesis from dataflow graphs," in *Proc. Int. Symp. Syst. Synthesis*, 1998.
[25] S. Ritz, M. Pankert, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Proc. Int. Conf. Appl.-Specific Array Processors*, Oct. 1993.
[26] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, "FunState—An internal representation for codesign," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 1999.
[27] D. J. Wang and Y. H. Hu, "Fully static multiprocessor array realizability criteria for real-time recurrent DSP applications," *IEEE Trans. Signal Processing*, vol. 42, pp. 1288–1292, May 1994.
[28] P. Wauters, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-dynamic dataflow," in *Proc. Fourth EUROMICRO Workshop Parallel Distrib. Process.*, Jan. 1996.
[29] E. Zitzler, J. Teich, and S. S. Bhattacharyya, "Evolutionary algorithms for the synthesis of embedded software," *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, vol. 8, pp. 452–455, Aug. 2000.

Bishnupriya Bhattacharya received the the B.S. degree in computer science from Jadavpur University, India, and the M.S. degree from the Department of Electrical and Computer Engineering from the University of Maryland, College Park.

She is a research engineer with Cadence Design Systems Inc., San Jose, CA. Her research interests focus on modeling, simulation, and synthesis through system-level CAD tools for embedded applications, especially in the dataflow model of computation. Her work has been published and recognized in prestigious conferences, as well as in the industry.

Shuvra S. Bhattacharyya (SM'01) received the B.S. degree from the University of Wisconsin, Madison, and the Ph.D. degree from the University of California, Berkeley.

He is an Associate Professor with the Department of Electrical and Computer Engineering and the Institute for Advanced Computer Studies, University of Maryland, College Park. His research interests center around architectures and computer-aided design for embedded systems, with emphasis on hardware/software codesign for digital signal processing. He has held industrial positions as a Researcher at Hitachi and as a Compiler Developer at Kuck & Associates. He consults for industry in the areas of compiler techniques and multiprocessor architectures for embedded systems. He is the coauthor of two books and the author or coauthor of more than 40 refereed technical articles.

Dr. Bhattacharyya received the NSF Career Award.