In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing.
Istanbul, Turkey, June 2000.

PARAMETERIZED DATAFLOW MODELING OF DSP SYSTEMS

Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya

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

ABSTRACT[1]

Dataflow has proven to be an attractive computation model for programming 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 [3]. 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 an arbitrary 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, CSDF, and SSDF. In this paper, we develop a precise, formal semantics for *parameterized synchronous dataflow* that allows data-dependent dynamic DSP systems to be modeled in a natural and intuitive fashion. Desirable properties of a modeling environment like dynamic re-configurability and design re-use emerge as inherent characteristics of the parameterized framework. An example of a speech compression application is used to illustrate the efficacy of the parameterized modeling techniques in real-life data-dependent DSP systems.

1. MOTIVATION

Algorithms for digital signal processing are often most naturally described by block diagrams in which computational blocks (actors) are interconnected by links (edges) that represent sequences of data values. In the synchronous dataflow (SDF) model [10] the number of data values (tokens) produced and consumed by each actor is fixed and known at compile time. Fig. 1(a) shows a simple SDF graph. The symbols adjacent to the inputs and outputs of the actors represent the numbers of tokens consumed and produced. The primary benefits that SDF provides are static scheduling and a high degree of compile-time predictability. Although an important class of useful DSP applications can be modeled efficiently in SDF [3, 10], its expressive power is limited to fully 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, and possibly, allowing representations that expose more optimization opportunities to a compiler. In cyclo-static dataflow (CSDF) [4], token production and consumption can vary between actor executions, as long as the variation forms a certain type of periodic pattern. Lee has proposed the multidimensional dataflow (MD-SDF) [9] model to efficiently handle multidimensional data. In scalable synchronous dataflow (SSDF) [12], each SSDF actor has the capacity to process any integer multiple of the basic token production/consumption quantities at an output/input port, leading to reduced inter-actor context-switching in synthesized implementations. The Boolean dataflow (BDF) [5] model allows non-SDF actors, where 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. This significantly increases the expressivity over SDF — indeed BDF is Turing-complete. Gao et al. have studied well-behaved stream flow (WBSF) [6], where non-SDF actors (*switch* and *merge*) are only used as a part of two predefined schemas. CSDF, MD-SDF, SSDF, and WBSF all possess a high degree of compile-time predictability, be extending SDF techniques. In BDF key verification issues are undecidable, and run-time scheduling is necessary in some cases.

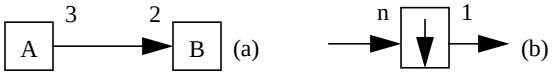

Figure 1. Dataflow actors and edges. (a) An SDF graph. (b) A PSDF downsampler actor, that decimates by a factor of n.

1. This work was sponsored by Northrop Grumman and NSF (CAREER MIP9734275).

In this paper, we propose *parameterized dataflow* as a meta-modeling technique that can be applied to a wide range of dataflow models to increase their expressive power. Parameterized dataflow imposes a hierarchy discipline, and requires that certain properties hold for each hierarchical subsystem over well-defined windows of time. Our parameterization concepts can be incorporated into any dataflow model in which there is a notion of a graph *iteration*. For example, a *minimal periodic schedule* [3] can be a natural notion of an iteration in SDF, SSDF, CSDF, and MD-SDF. Similarly, in BDF, a *complete cycle* [5], when it is exists, can be used to specify an iteration of a subsystem. For clarity and uniformity, and because SDF is currently the most popular dataflow model for DSP, in this paper, we have developed parameterized dataflow formally in the context of SDF. To demonstrate extensibility to other dataflow models, [1] provides a PCSDF model of the speech compression application given in Sec. 3.

In a broad sense, our goal of enhancing the effectiveness of a range of existing and conceivable models of computation is in the same spirit as *charts (starcharts) [7] — a meta-modeling technique that allows hierarchical finite state machines to be embedded in a variety of concurrency models, including dataflow, discrete event, and synchronous/reactive. Parameterized dataflow is different in that it does not require any departure from the dataflow domain, which may be advantageous for users of DSP design tools who are accustomed to working with the dataflow model.

The rest of this paper is constructed as follows. In Section 2 we present the formal semantics of parameterized synchronous dataflow (PSDF), and mention scheduling techniques. Section 3 shows an example of a speech compression application modeled in PSDF. Finally, we conclude in Section 4 with a summary of parameterized dataflow and directions for further work.

2. PARAMETERIZED SYNCHRONOUS DATAFLOW

In this section we present a brief overview of the formal semantics of the PSDF model. Further details on the concepts introduced here can be found in [1].

2.1 PSDF Graphs

A *PSDF graph* is composed of *PSDF actors* and *PSDF edges*. A PSDF actor A 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) at its input and output *ports*. e.g., a PSDF downsampler actor can be characterized by two parameters {*factor*, *phase*}, where *factor* represents the decimation factor, and *phase* denotes the index of the input token transferred to the output. The functionality of the downsampler actor depends on both its parameters, while the dataflow behavior depends only on the *factor* parameter.

An application designer determines a *configuration* of a PSDF actor A (denoted $config_A$), by assigning values to the parameters of A, where each parameter p is either assigned a value from an associated set, called $domain(p)$, or is left unspecified. These statically unspecified parameters are assigned values at run-time, that can change dynamically, thus dynamically modifying the actor behavior.

Like a PSDF actor, a *PSDF edge* e also has an associated parameterization, and configuration ($config_e$), where the delay characteristics on an edge (the number of units of delay on the edge, initial values of the delay tokens, and the delay *re-initialization period*) can in general depend on its parameter configuration. In addition, to facilitate practical implementations, an upper bound is specified on the number of tokens produced and consumed onto the edge, and the number of delay tokens residing on the edge, whenever these quantities are left statically unspecified. This appears similar to bounded dynamic dataflow (BDDF) [11], where dynamic dataflow is modeled through dynamic actor ports that can have data-dependent token transfer at run-time, bounded by a specified maximum value. However, our use is different in the sense that for BDDF, the token transfer bounds do not always guarantee bounded memory, since it does not possess well-defined local regions of well-behaved operation, whereas the PSDF operational semantics guarantees bounded memory execution for consistent specifications, as we will discuss in Sec. 2.4.

In a PSDF graph G, edges connect a set of actor output ports to a set of actor input ports. There can exist actor ports, called the *interface ports* of G, on which no edges in G are incident. All the statically unspecified actor/edge parameters in G propagate "upwards" as parameters of G, denoted $params(G)$. Clearly then, given a configuration C of the parameters of G, a pure SDF graph, denoted $instance_G(C)$, emerges by "applying" the configuration C to unspecified parameters in G.

2.2 PSDF Specifications

A DSP application will usually be modeled in PSDF through a *PSDF specification*, also called a *PSDF subsystem*. A dominant idea in the architecture of our parameterized dataflow framework is the decomposition of a specification Φ into three distinct graphs — the *init graph* Φ_i, the *subinit graph* Φ_s, and the *body graph* Φ_b. The body graph models dataflow behavior of the specification, while the init and subinit graphs control the behavior of the body graph by configuring the parameters of the body graph.

When a PSDF specification Φ is embedded within a "parent" graph through a *hierarchical PSDF actor*, in general, it can have dataflow communication with actors in the parent graph. Φ_i does not participate in this dataflow. Φ_s may only accept dataflow inputs at its interface input ports, while Φ_b may accept dataflow input and produce dataflow output at its interface ports. The interface output ports of Φ_i and Φ_s are reserved exclusively for configuring parameter values. All the parameters of Φ_b are configured at the interface outputs of Φ_i and Φ_s. Each parameter of Φ_s is either configured at an interface output of Φ_i, or is bound to a dataflow input port of Φ (i.e. the value of the input token at this port is set 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, denoted $params(\Phi)$. These specification parameters are set by the init and subinit graphs of hierarchically higher-level subsystems.

Intuitively, the execution phase of every PSDF subsystem is preceded by a configuration phase. The invocation semantics specifies 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. Thus, parameter values of Φ_b and Φ_s that are configured in Φ_i remain constant throughout an execution of the parent graph, while parameter values of Φ_b that are configured in Φ_s remain constant throughout each invocation of Φ, but can change across invocations.

The concept of actor parameters has 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 re-configuring the behavior of dataflow actors, edges, graphs, and subsystems.

2.3 Consistency in PSDF

Consistency issues in PSDF can be classified into *dataflow consistency*, which arises as a part of the underlying SDF model, and *local synchrony consistency*, which is a direct consequence of the hierarchical parameterized representation that PSDF proposes. For space constraints, we provide only an intuitive idea here. Precise definitions can be found in [1]. Given a PSDF graph G, if for every possible configuration C that can be applied to G, the instantiated SDF graph $instance_G(C)$ has a *valid schedule* (i.e. it is *sample rate consistent* and is *deadlock*-free) [3], and the token transfer and delay bounds are satisfied for every edge in G, then G is defined to be *inherently dataflow consistent* (or simply *dataflow consistent*). If every configuration C of G leads to an invalid schedule for $instance_G(C)$ or violates the upper bounds on at least one edge in G, then G is termed *dataflow inconsistent*. If G is neither inherently dataflow consistent, nor dataflow inconsistent, then it is *partially dataflow consistent*. Intuitively, a PSDF specification Φ is locally synchronous if Φ_i, Φ_s and Φ_b are dataflow consistent and the interface dataflow behavior of Φ (token production and consumption at interface ports) is determined entirely by the init graph Φ_i. Four conditions need to be satisfied for the latter. The first two conditions called the *unit transfer* conditions are: Φ_i and Φ_s must produce exactly one token at each of their output ports on each invocation. The third and fourth conditions require that the number of tokens consumed by Φ_s from each of its interface input ports must be a function of the subinit graph parameters that are configured by Φ_i, and the number of tokens produced or consumed at each interface port of Φ_b must be a function of the body graph parameters that are configured by Φ_i. Similar to the classifications of dataflow consistency, a subsystem Φ can also be classified as *inherently locally synchronous* (or simply *locally synchronous*), *locally non-synchronous*, and *partially locally synchronous* [1]. In association with the parameter configuration mechanism and the subsystem invocation semantics, the local synchrony conditions ensure that a hierarchically nested PSDF specification behaves like an SDF actor throughout any sin-

```
function call(G, C)
  foreach hierarchical actor representing subsystem Φ in G
    C_{Φ,i} = configure_graph(Φ_i, C)
    C_{Φ,init} = call(Φ_i, C_{Φ,i})
    C_{Φ,s} = configure_graph(Φ_s, C ∪ C_{Φ,init})
    C_{Φ,b} = configure_graph(Φ_b, C ∪ C_{Φ,init})
    precompute_interface_tokens(Φ_s, C_{Φ,s}, Φ_b, C_{Φ,b})
  end for
  foreach leaf actor A in G
    apply_configuration(C, config_A)
  foreach edge e in G apply_configuration(C, config_e)
  compute_SDF_repetitions_vector(G)
  S = compute_SDF_schedule(G); C_i = ∅; C_o = ∅
  while ((L = get_next_firing(S)) ≠ NULL)
    if (L is a hierarchical actor representing subsystem Φ)
      C_{Φ,s} = configure_graph(Φ_s, C ∪ C_{Φ,init} ∪ C_i)
      C_{Φ,subinit} = call(Φ_s, C_{Φ,s})
      C_{Φ,b} = configure_graph(Φ_b, C_{Φ,init} ∪ C_{Φ,subinit})
      call(Φ_b, C_{Φ,b})
      verify_local_synchrony(Φ)
    else execute L
      if (L sets parameter p to value v at output port θ)
        update_either_configuration(C_o, C_i, θ, {p, v})
    end if
  return C_o
end function
```

Figure 2. The operational semantics of PSDF.

gle invocation of its parent graph, which is necessary for schedulability of a PSDF graph.

Thus, similar to BDF, and BDDF, consistency of a PSDF specification is dependent on the input sequences that are applied to it, which rules out PSDF as a *binary-consistency* model [2] in the lines of SDF, CSDF, SSDF and WBSF, where every specification is either consistent or inconsistent, and there are no partially consistent specifications. Binary consistency is convenient from a verification viewpoint, but translates to restricted expressivity.

2.4 PSDF Operational Semantics and Scheduling

Based on the formalism discussed in the previous sections, a precise operational semantics for PSDF is given in Fig. 2. For space constraints, we have used a pseudo-code format. The figure shows the routine call( ), which given a 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 consistency. The routine returns the output configuration C_o determined for those parameters that are set at the interface output ports of G. Execution of a PSDF specification Φ is initiated by invoking the routine call( ) on the graph G in which Φ is implicitly assumed to be embedded, with an empty configuration C.

Among the verification tasks, verifying dataflow consistency of G and unit transfer consistency of a subsystem Φ embedded in G is straightforward [1]. Verifying the third and fourth conditions for local synchrony of subsystem Φ consists of pre-computing the token flow at the interface ports of Φ_s and Φ_b after firing Φ_i, and subsequently comparing the pre-computed value with the actual token flow obtained after firing Φ_s and Φ_b. While pre-computing the token flow, parameters of Φ_s and Φ_b that are not configured in Φ_i have an unknown value, and the configure_graph( ) routine assigns *default values* (as specified by the application programmer) for these parameters. Thus, parameters of Φ_s and Φ_b that are not set up in Φ_i must have default values specified judiciously, such that their inter-relationship in determining the interface token flow of Φ is the same as any combination of values that these parameters can take on at run-time.

As outline here, the PSDF operational semantics performs precise run-time consistency verification, as opposed to existing non-binary-consistency models like BDF, and BDDF, which do not provide comparable, well-defined, inconsistency detection techniques, in general, even at run-time. In particular, an inconsistent input in PSDF will eventually be detected as being inconsistent, based on a well-defined semantical criterion. Both inherent and partial local synchrony of PSDF specifications ensure bounded memory requirements throughout execution of the system, as a sequence of consistent SDF executions. Thus, in general, the parameterized framework applied on a binary-consistency model produces a non-binary-consistency model that trades off expressive power with compile-time predictability, and achieves improved levels of run-time predictability.

In addition, it is possible to streamline the implementation of the PSDF operational semantics by careful compile-time analysis. Indeed, the PSDF model provides a promising framework for productive compile-time verification that warrants further investigation. As an example of such streamlining, we have developed an efficient *quasi-static* scheduling algorithm, based on a *clustering* technique [3] for a class of PSDF specifications. Our quasi-static schedules are generated at compile-time, but may contain code that performs data-dependent computations at run-time. Fig. 4 shows an example of a PSDF quasi-static schedule. Details of our scheduling algorithms can be found in [1]. We have implemented a tool that accepts a PSDF specification, and generates either a quasi-static or a run-time schedule for it, as appropriate.

3. A PSDF APPLICATION

In this section, we model a speech compression application as a PSDF specification and provide a quasi-static schedule for it. In Sections 2.1 and 2.2, 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), 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 [1]. An example is shown in Fig. 1(b), which models a downsampler actor in PSDF, decimating by a factor of n. The parameter n is a subsystem parameter in this case, and the actor parameter *factor* has to be assigned the subsystem parameter value n. For the PSDF examples shown in this paper, the token flow at an actor port is indicated either as a static integer, or as a symbolic expression of subsystem parameters.

Fig. 3 shows a PSDF subsystem *Compress* modeling a speech compression application. Dashed rectangles are used to enclose PSDF graphs. In the application, 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 [8]. In the init graph, the *genHdr* actor generates a stream of header packets, where each header contains information about a 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* actor accepts small speech

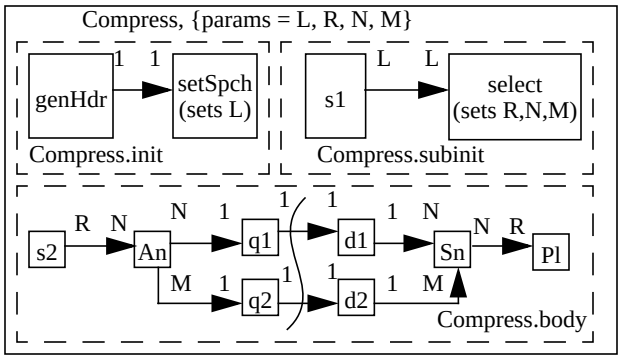

Figure 3. A speech compression application modeled in PSDF.

repeat 5 times **{ fire** *genHdr*; **fire** *setSpch;* **/*** sets L ***/**
fire *s1;* **fire** *select* **/*** sets R, N, M ***/**
fire *s2;* **repeat** (R/N) times **{ fire** *An*
repeat (N) times **{** fire *q1*; fire *d1* **}**
repeat (M) times **{** fire *q2*; fire *d2* **}; fire** *Sn* **} fire** *Pl*
}

Figure 4. A PSDF quasi-static schedule for Fig. 3.

segments of size N, and performs linear prediction, producing M AR coefficients and the residual error signal of length N at its output. These are quantized, encoded (actors *q1*, *q2*), and transmitted to the receiver side, where these are dequantized (actors *d1*, *d2*) and each segment is reconstructed in the *Sn* actor through AR modeling. Finally the *Pl* actor plays the entire reconstructed speech instance. The curved line in the figure demarcates the transmitter from the receiver. 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 in order to achieve high compression ratios. N and M, along with the zero-padded speech sample length R are modeled as subsystem parameters of *Compress*, that are set in the subinit graph, where the *select* actor reads the entire speech instance, and examines it to determine N and M, using any of the existing techniques [8]. R is computed such that the segment size exactly divides the zero-padded length.

Our purpose here is to provide an overview of the design process, using mixed-grain DSP actors, such that PSDF-specific aspects of the model are emphasized. Due to space constraints, upper bounds on the edges have not been indicated in the specification and in the schedule. We have also omitted all actor parameters and all subsystem parameters that do not affect the system's dataflow behavior, e.g., *s1* and *s2* will have actor parameters indicating the length of the speech signal to be generated (set to L and R respectively). A detailed explanation is available in [1], which also documents an alternate PSDF specification, where a speech instance is generated only once, instead of twice as in this case. More DSP applications, modeled in PSDF can be found in [1].

Fig. 4 shows the quasi-static schedule obtained for the *Compress* specification, assuming that the scheduler knows N exactly divides R. The application is run five times, and in each run a different speech instance can be processed. Thus it is clear that parameterized modeling naturally allows the same basic design to be re-configured with appropriate parameter values so as to process different sets of inputs in different ways, leading to more flexible dataflow behavior and increased design re-use.

4. CONCLUSIONS AND FUTURE WORK

We have introduced a parameterized dataflow framework that can be applied as a meta-modeling technique to a wide range of dataflow models to significantly increase their expressive power. A formal semantics for parameterized synchronous dataflow has been presented. An example of a speech compression application has been used to illustrate PSDF concepts.

Parameterization of subsystem functionality comes out 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, it makes PSDF a natural and intuitive choice for modeling data-dependent, dynamic DSP systems. The parameterized framework also supports increased design modularity, leading to condensed actor libraries for block-diagram DSP programming environments in some cases [1]. PSDF possesses a robust and elegant operational semantics, and efficient quasi-static schedules can be developed for a large class of PSDF specifications, making it attractive for synthesis purposes. Formal verification of PSDF specifications and optimized synthesis of quasi-static schedules appears to be promising directions for future work. Incorporating dynamically re-configurable *graph topologies* into the parameterization framework also appears to be a feasible concept, thus further increasing its expressive power.

5. REFERENCES*

[1] B. Bhattacharya, S.S. Bhattacharyya, "Parameterized Modeling and Scheduling of Dataflow Graphs", Tech. Report UMIACS-TR-99-73, Institute for Advanced Computer Studies, University of Maryland, College Park, November, 1999.

[2] S. S. Bhattacharyya, R. Leupers, P. Marwedel, "Software Synthesis and Code Generation for Signal Processing Systems", Tech. Report UMIACS-TR-99-57, Institute for Advanced Computer Studies, University of Maryland, College Park, September, 1999.

[3] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[4] G. Bilsen, M. Engels, R. Lauwereins, J. A. Peperstraete, "Cyclo-static Data Flow", *Proc. ICASSP*, 1995.

[5] J. T. Buck, E. A. Lee, "Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model", *Proc. ICASSP*, 1993.

[6] G. R. Gao, R. Govindarajan, P. Panangaden, "Well-Behaved Dataflow Programs for DSP Computation", *Proc. ICASSP*, 1992.

[7] A. Girault, B. Lee, E.A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models", October 19, 1998, revised from Memorandum UCB/ERL M97/57, Electronics Research Laboratory, UC, Berkeley, August 1997.

[8] S. Haykin, *Adaptive Filter Theory*, third edition, Prentice Hall Information and System Sciences Series, 1996.

[9] E. A. Lee, "Representing and Exploiting Data Parallelism Using Multidimensional Dataflow Diagrams", *Proc. ICASSP*, 1993.

[10] E.A. Lee, D.G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, February, 1987.

[11] M. Pankert, O. Mauss, S. Ritz, H. Meyr, "Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis", *Proc. ICASSP*, 1994.

[12] S. Ritz, M. Pankert, H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs", *Proc. International Conference on Application-Specific Array Processors*, 1993.

* References [1, 2] are available by anonymous ftp from *dspserv.eng.umd.edu* in the directory *pub/dspcad/papers*.