B. Bhattacharya. "Parameterized modeling and scheduling for dataflow graphs."
Master's thesis, Department of Electrical and Computer Engineering,
University of Maryland, College Park, November 1999.

PARAMETERIZED MODELING AND SCHEDULING

FOR DATAFLOW GRAPHS

Degree Candidate: Bishnupriya Bhattacharya

Degree and year: Master of Science, 1999

Thesis directed by: Dr. Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies

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. Many extensions to synchronous dataflow have been proposed to increase its expressivity, while maintaining its compile-time predictability properties as much as possible.

This thesis introduces 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. The parameterized dataflow framework is compatible with many of the existing dataflow models for DSP including SDF, CSDF, MD-SDF, and SSDF. For clarity, and because SDF is currently the most popular model for designing DSP systems, a precise, formal semantics for parameterized synchronous dataflow (PSDF) has been developed. PSDF 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. Scheduling techniques are presented for generating efficient quasi-static schedules for a class of PSDF specifications, geared towards software synthesis in embedded systems. Practical DSP applications are used to illustrate the efficacy of the parameterized modeling and quasi-static scheduling techniques in real-life data-dependent DSP systems.

PARAMETERIZED MODELING AND SCHEDULING FOR DATAFLOW GRAPHS

by

Bishnupriya Bhattacharya

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
1999

Advisory Committee:

Dr. Shuvra S. Bhattacharyya, Chair

Professor KouJuey Ray Liu

Professor Kazuo Nakajima

Chapter 1. Introduction

In the area of digital signal processing (DSP), dataflow is widely recognized as a natural model for specifying DSP applications. In dataflow, a program is represented as a directed graph, called a *dataflow graph*, in which vertices, called *actors*, represent computations and edges represent FIFO channels, (also called *buffers*). These channels queue data values, in the form of *tokens*, which are passed 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.

Different models have been proposed within the dataflow programming framework, based on diverse objectives. One common strain that runs through these models is the trade-off between expressivity and compile-time predictability of specifications expressed in them. Increased predictability translates to desirable features such as more thorough optimization and verification, and lower run-time overhead for scheduling and memory management, but at the cost of sacrificing some expressive power. Lee and Messerschmitt have proposed the synchronous dataflow (SDF) model [26], which is a restricted version of dataflow, where the number of tokens produced (or consumed) by an actor firing, on each output (or input) is a fixed number that is known at compile time. With relatively low expressivity but very high compile-time predictability, SDF leans heavily towards one end of the

spectrum from the expressivity/predictability trade-off perspective. At the other end of the spectrum we have dynamic dataflow (DDF) which supports arbitrary data-dependent behavior using non-SDF actors with unknown token production/consumption (at compile-time) that can be used to model conditionals, iterations, and recursion. DDF possesses high expressivity — it is Turing complete, but permits a minimum of useful compile-time analysis and optimization. An implementation of a DDF based development environment is available in Ptolemy [12] — a tool that provides an object-oriented framework for simulation, prototyping and software synthesis of heterogeneous systems.

Many extensions to the SDF model have been proposed that lie somewhere in-between DDF and SDF within the spectrum. The objective is to broaden the range of applications that can be represented vis. a vis. SDF, while maintaining its compile-time predictability properties as much as possible, and at the same time, allowing representations that expose more optimization opportunities to a compiler. Multidimensional dataflow (MD-SDF) [24] and cyclo-static dataflow (CSDF) [15] lie in the latter category. MD-SDF allows efficient modeling of multidimensional applications, and exposes parallelism more effectively than pure SDF. It also addresses the issue of re-initialization of delays on graph edges, which allows certain applications that cannot be represented in SDF to be modeled in MD-SDF. In CSDF token production and consumption can vary between actor firings as long as the variation forms a certain type of periodic pattern. 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 like constant propagation and dead code elimination [7]. Yet another advantage offered by CSDF is that while hierarchically abstracting the functionality of an SDF graph by representation through a single actor, the SDF model may introduce deadlock in a system specification, which can often be avoided if the functionally equivalent CSDF actor is used [7]. Thus MD-SDF and CSDF possess increased expressive power over SDF, with similar compile-time predictability properties.

Well-behaved stream flow graphs (WBSFG) proposed by Gao et al. [17], allows the use of two non-SDF dynamic actors (*switch* and *select*) for modeling conditionals and data-dependent iteration, but in a restricted fashion such that the model retains the key predictability properties of SDF, while offering increased expressivity. On the other hand, in the Boolean dataflow model (BDF), developed by Buck [10], these kinds of restrictions are not present, and hence it has greater expressive power — indeed it is Turing-complete. Consequently, it does not have guaranteed compile-time predictability properties. The objective in BDF is to extend SDF techniques to generate *quasi-static schedules* [25] whenever possible, and fall back on fully dynamic scheduling and analysis otherwise. In quasi-static scheduling some actor firing decisions are made at run-time, but only where absolutely necessary.

The cyclo-dynamic dataflow model (CDDF) [35] extends cyclo-static dataflow in such a manner that all BDF and CSDF graphs can be expressed in the CDDF model. In addition, it allows the user to convey application-specific knowledge

about the internals of an actor to the compiler which can lead to better analyzability than the BDF model [35].

Additionally, some models of computation have been proposed that combine the SDF paradigm with finite state machine (FSM) models. Two examples of such composite modeling approaches are heterochronous dataflow (HDF) [18], and bounded dynamic dataflow (BDDF) [28]). By modeling control-flow along with dataflow, these approaches lead to increased expressivity over SDF alone. In Sections 1.2 and 1.3 we take a more detailed look at synchronous dataflow, and various extensions of the SDF model.

1.1. Basic Notation

We denote the set of positive integers by the symbol Z^+, the set of *extended positive integers* $(Z^+ \cup \{\infty\})$ by $\overline{Z}$; and the set of natural numbers $\{0, 1, 2, \ldots\}$ by $\aleph$. The greatest common divisor of two integers a and b is denoted by $gcd(a, b)$. The remainder obtained by dividing an integer a by an integer b is denoted as $mod(a, b)$. The notation $g : D \rightarrow R$ represents a function g whose domain and range are D and R, respectively. The *image* of $D' \subseteq D$ *under* g, denoted $g(D')$, is defined by $g(D') = \{g(x) | x \in D'\}$, and $g(D)$ is called the *image* of g. The *cardinality* of a finite set S, denoted $|S|$, simply specifies the number of elements in S. The symbol – is used to denote *set difference*.

A *directed multigraph* G denotes an ordered pair (V, E), where V and E are finite sets, and associated with each $e \in E$ there are two properties $src(e)$ and $snk(e)$ such that $src(e), snk(e) \in V$. Each member of V is called a *vertex* of G

and each member of E is called an *edge*. $src(e)$ is called the source vertex of e, and $snk(e)$ is called the sink vertex of e. A *subgraph* associated with any $V' \subseteq V$ is the directed multigraph formed by V' together with the set of edges $\{e \in E | (src(e), snk(e) \in V')\}$, and is denoted by $subgraph(V', G)$; if G is understood from context, we may simply say $subgraph(V')$. A *path* in (V, E) is a nonempty sequence $e_1, e_2, e_3, \ldots \in E$ such that $snk(e_1) = src(e_2)$, $snk(e_2) = src(e_3)$, and so on. Given a finite path $p = e_1, e_2, \ldots, e_n$, we say that p is directed from $src(e_1)$ to $snk(e_n)$. A path that is directed from some vertex to itself is called a *cycle* or a *directed cycle*, and a *fundamental cycle* is a cycle of which no proper subsequence is a cycle. A graph is called *acyclic*, if it does not contain any cycles, and it is called *cyclic* otherwise. G is called a *bipartite graph* if its vertex set V can be decomposed into two disjoint subsets V_1 and V_2 such that every edge in G joins a vertex in V_1 with a vertex in V_2.

1.2. Synchronous Dataflow (SDF)

Synchronous dataflow is a restricted version of dataflow in which the number of tokens produced (or consumed) by an actor firing on each output (or input) is a fixed number that is known at compile time. Each edge in an SDF graph also has a non-negative integer delay associated with it, which corresponds to the number of initial tokens on the edge. If for each edge in the graph, the number of tokens consumed, and the number of tokens produced equal one, then we have *homogeneous* SDF. In *single-rate* SDF, there is no production/consumption mismatch across an edge; i.e. the number of tokens produced is equal to the number of tokens consumed

for every edge, and thus all actors are invoked at the same average rate. On the other hand, *multi-rate* SDF allows sample rate mismatches across edges.

Fig. 1 shows a multi-rate 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. A delay is also indicated by a triangular mark on an edge. 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 $d(e)$. Also, $p(e)$ and $c(e)$ denote the number of tokens produced onto e by $src(e)$ and consumed from e by $snk(e)$.

A *firing* of an actor in G corresponds to removing $c(\alpha)$ tokens from the head of the buffer for each input edge α, and appending $p(\alpha)$ tokens to the buffer for each output edge β. We say that an actor is *fireable* if there are enough input tokens on each input buffer to fire that actor. A *schedule* for G is a sequence $L = f_1 f_2 f_3 \ldots$, which can be finite or infinite, of actors in G. Each term f_i of this sequence is called an *invocation* of the corresponding actor in the schedule. For each i, f_i is said to be an *admissible firing* if it is fireable immediately after $f_1, \ldots, f_{i-1}$ have fired in succession. The schedule L is an *admissible schedule* for G if f_i is an admissible firing for each i. The process of successively firing the invocations in an

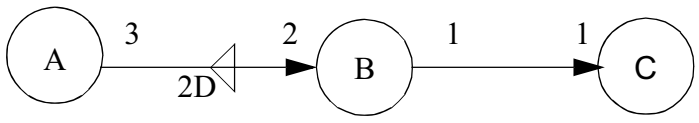


Figure 1. A simple SDF graph.

admissible schedule is called *executing* the schedule, and if a schedule is executed repeatedly, each repetition of the schedule is called a *schedule period* of the execution.

As we fire the invocations in the schedule, we can represent the *state* of the system by the numbers of tokens queued on the buffers associated with the edges. A finite schedule L is a *periodic schedule* if it invokes each actor at least once and produces no net change in the system state. The number of tokens queued on each edge is left unchanged. Schedules that are both periodic and admissible are referred to as *valid* schedules. An SDF graph is *consistent* if and only if it has a valid schedule.

In DSP applications, we are mostly dealing with infinitely or indefinitely long sequences of input data, and thus it is mandatory that we support infinite schedules of actor executions. But there are several issues related to such infinite schedules:

- The buffer memory requirement along each edge may become unbounded;
- The graph may become deadlocked;
- The schedule may or may not be periodic;

The most significant advantage of SDF is that all of these issues can be resolved at compile-time. There exist efficient techniques to determine at compile-time whether or not an arbitrary SDF graph has a valid schedule (a periodic schedule that neither deadlocks nor requires unbounded buffer sizes), and to construct a valid schedule such that the resulting target program is optimized [8]. The minimum number of

times an actor has to be fired in a valid schedule is represented by a vector $\mathbf{q}_G$, indexed by the actors in G (we often suppress the subscript G if it is understood). These minimum numbers of firings can be derived by finding the minimum positive integer solution to the *balance equations* for G, which specify that $\mathbf{q}$ must satisfy

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

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

Another important benefit that the static properties of SDF offer is potential for thorough optimization, and effective optimization techniques have been developed in the contexts of improving the efficiency of buffering code [6], data memory minimization [1], joint minimization of code and data [8, 32, 36], high-throughput block processing [33], multiprocessor scheduling (there have been numerous efforts in this category — for example, see [29, 3, 13, 22, 30]), synchronization optimization [9], as well as a variety of other objectives.

In this thesis, we will be looking at scheduling techniques for SDF graphs, and their extensions. Fig. 2 shows a commonly used model for compiling an SDF graph. The compilation begins with constructing a periodic schedule from the SDF graph. A *threading compiler* steps through this schedule and for each actor instance that it encounters, it generates a block of code, derived from a predefined library of actor code blocks, that implements the actor. Typically when defining a new actor, the user will specify the code to implement that actor. The sequence of code blocks

output by the code generator is processed by a *storage allocation* phase that inserts the necessary code to route data appropriately between actors and assigns variables to memory locations. The output of this storage allocation phase is the target program. Further details on this method of compilation can be found in [8].

1.3. Extensions to the SDF Model

It is possible to describe a large class of useful DSP applications in SDF, which provides the benefits of static scheduling, as described in Section 1.2. However, many interesting applications also require some amount of dynamic or data-dependent behavior, at the inter-actor level, which is not allowed in the SDF model. Thus, many extensions to the SDF model have been proposed, where the objective

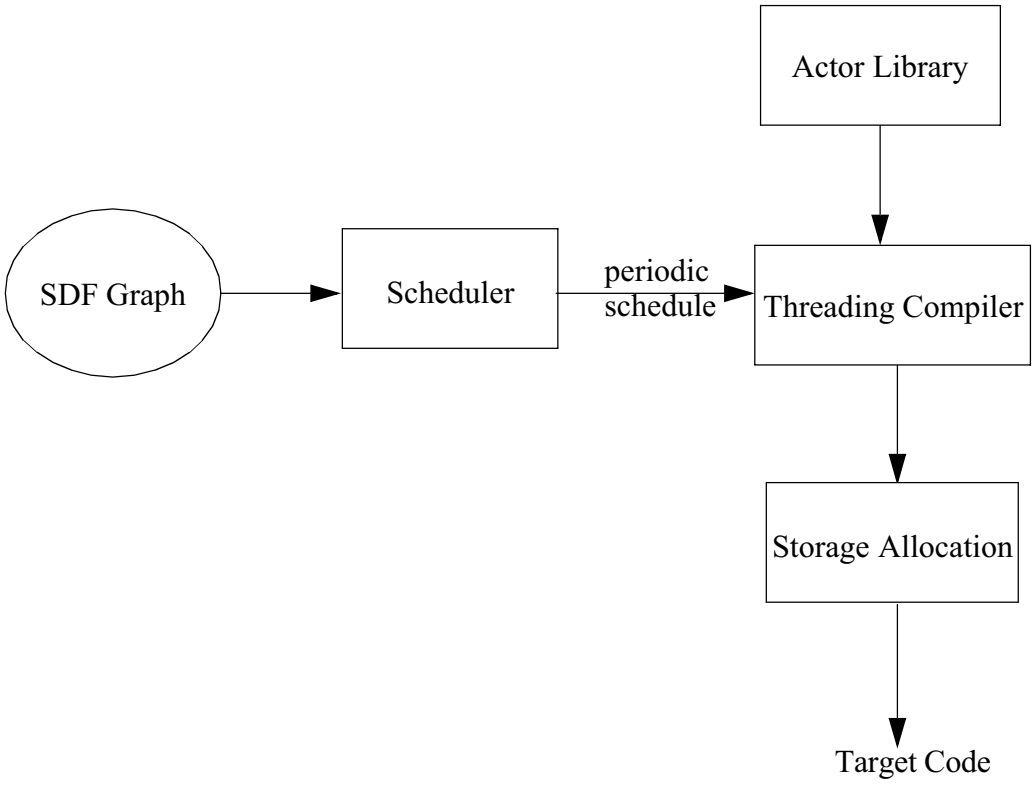


Figure 2. Compiling an SDF graph.

is to accommodate a broader range of applications, while maintaining a significant part of the compile-time predictability of SDF.

1.3.1 Boolean Dataflow

In the Boolean dataflow model, developed by Buck [10], 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. Thus a control token transferred by a control port (input or output of an actor) controls the number of tokens transferred by a conditional port.

BDF extends the analysis techniques used for SDF graphs to handle BDF actors with conditional ports, by associating symbolic expressions with conditional ports [10]. In Fig. 3, the *switch* actor is shown with its symbolic annotations. The *switch* actor is a BDF actor that reads one token from the control input, and depending on whether the value of the control token is true or false, routes the input to either the output marked T, or the output marked F. One possible interpretation of the symbolic annotations shown in the figure is: given a sequence of n actor executions of the *switch* actor, in which the proportion of TRUE Boolean tokens con-

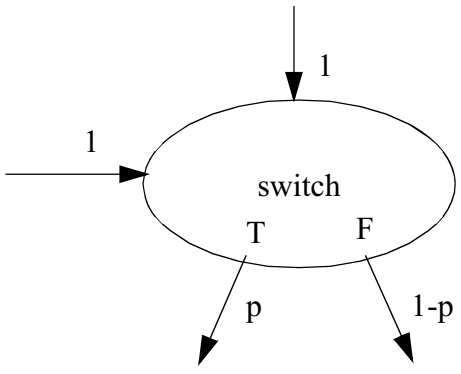


Figure 3. The *switch* actor in the Boolean Dataflow model.

sumed by the control port is p, the number of tokens produced on the TRUE output of the *switch* actor is np, and the number of tokens produced on the FALSE output is $n(1-p)$. A *topology matrix* [8] can now be constructed, similar to SDF graphs. Given a connected BDF graph G, a topology matrix Γ of G is a matrix whose rows are indexed by the edges in G and whose columns are indexed by the actors in G, and whose entries are defined by

$$\Gamma(\alpha, A) = \begin{cases} p(\alpha), \text{ if } A = src(\alpha) \\ -c(\alpha), \text{ if } A = snk(\alpha) \\ 0, \text{ otherwise} \end{cases} \quad (2)$$

The topology matrix will not be a constant, as in SDF, rather it will be a function of the symbolic variables present in the BDF graph. The topology matrix is a compact matrix-vector form of representing the balance equations that we discussed in Section 1.2 (1). For boolean dataflow, it is possible to solve these balance equations symbolically. This symbolic solution can lead to the detection of a *complete cycle*, which is a sequence of actor executions that returns the BDF graph to its original state (i.e. there is no net change in the number of tokens residing in the FIFO queue corresponding to each edge).

In constructing a schedule for BDF actors, Buck tries to come up with a quasi-static schedule, where each firing is annotated with the run-time condition under which the firing should occur. Buck has shown that the BDF model is Turing-complete, and hence many key decision problems, including that of finding a finite

complete cycle becomes undecidable for BDF. Thus, Buck presents heuristics for finding finite complete cycles in the form of a clustering algorithm, which attempts to map the graph into traditional control structures like *if-then-else* and *do-while*. If this clustering technique succeeds in reducing the graph to a single cluster, the graph is executed with the quasi-static schedule corresponding to the clusters. Otherwise, the resulting clusters are executed dynamically.

The BDF model has subsequently been extended by Buck to *Integer Controlled Dataflow* [11], where the value of the control tokens can be arbitrary integers instead of being Boolean. The scheduling techniques developed there are more or less extensions of those for the BDF model.

1.3.2 Well-Behaved Stream Flow

Gao et al. have studied a programming model called well-behaved stream flow (WBSF) [17], in which non-SDF actors (*switch* and *select*) are only used as a part of two predefined schemas called the *conditional schema*, and the *loop schema.* These restrictions guarantee that infinite schedules can be implemented with bounded memory. However, because of these very restrictions, Gao's model has much less expressive power than Buck's BDF model. In particular Gao's model is not Turing-complete.

1.3.3 Multidimensional Dataflow

Lee has proposed an extension of SDF to handle multidimensional data efficiently [24]. The standard SDF model assumes one dimensional data streams, and although a multidimensional stream can be embedded in a one dimensional stream,

the result is not very elegant, and does not expose data parallelism efficiently. In multidimensional SDF (MD-SDF), programming for data parallelism (in addition to functional parallelism that dataflow naturally models) in a graphical block diagram environment is a key objective.

Fig. 4(a) shows a simple 2D-SDF graph. The numbers of tokens produced and consumed are now given as M-tuples. Instead of one balance equation for an edge, there are now M balance equations — one for each dimension, which are solved for the smallest integers $r_{X,i}$. The solutions give the number of repetitions of actor X in each dimension i. For the graph in Fig. 4(a), the balance equations are given by $r_{A,1}O_{A,1} = r_{B,1}I_{B,1}$ and $r_{A,2}O_{A,2} = r_{B,2}I_{B,2}$. The *precedence graph* that can be automatically constructed from a MD-SDF graph exposes both functional and data parallelism leading to efficient scheduling.

MD-SDF also addresses the issue of re-setting of delays on an arc, which is necessary for correct functionality in certain kinds of applications involving repeated computations. Applications with such resetting of delays will be illustrated in Section 2.4. A delay in MD-SDF is coupled with a tuple, as shown in Fig. 4(b). It can be interpreted as specifying the boundary condition on the index space associ-

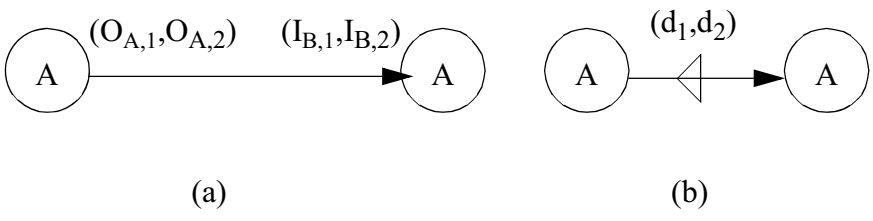


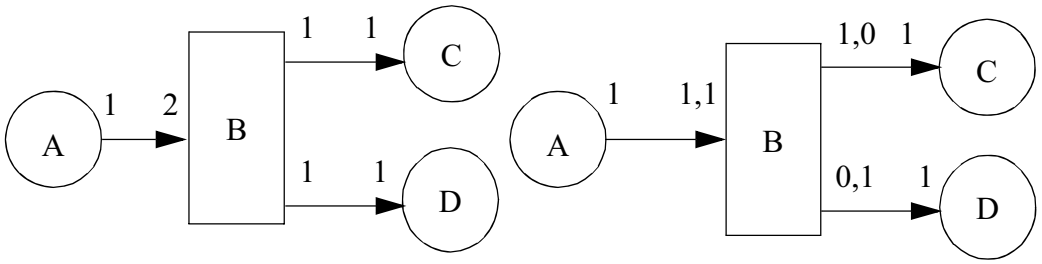
Figure 4. The Multidimensional Dataflow model. (a) A simple MD-SDF graph. (b) A multidimensional delay in MD-SDF.

ated with that arc. Thus for 2D-SDF as shown in the figure, it specifies the number of initial rows and columns.

1.3.4 Cyclo-Dynamic Dataflow

Cyclo-dynamic dataflow [35] is an extension of Cyclo-static dataflow (CSDF) [15]. In cyclo-static dataflow, the number of tokens produced and consumed by an actor can vary between firings, as long as the variations form a certain type of periodic pattern. Each time an actor is fired, a different piece of code called a *phase* is executed. CSDF specification of a *distributor* actor (actor *B*), which routes data received from a single input to each of two outputs in alternation, is shown in Fig. 5. Fig. 5(a) shows the SDF representation, while Fig. 5(b) shows the CSDF representation. A CSDF graph can be compiled as a cyclic pattern of pure SDF graphs, and static periodic schedules can be constructed in this manner. One advantage of CSDF over SDF is that it can sometimes lead to significantly lower buffer memory requirements.

Cyclo-dynamic dataflow extends cyclo-static dataflow, by introducing data-



Schedule: AABCD ABCABD

(a) (b)

Figure 5. The Cyclo-static Dataflow model compared with the Synchronous Dataflow model.

dependent flow of control in the CSDF model. The semantics are constructed such that the extra knowledge about the internals of the actors, which are known to the programmer, can be expressed in a natural way, and in a syntax that can be analyzed by automatic tools. The CDDF model allows the use of symbolic variables that reflects relevant properties of actor behavior. Similar to BDF, CDDF also has a concept of *control tokens*. A CDDF control token can determine the token transfer at an actor port, and the next actor phase to be executed. To allow the scheduler to evaluate the firing rule for a given phase, the model is restricted as follows: for input terminals the consumption numbers must not depend on a symbolic variable. Also, the actual phase that will be invoked by the scheduler must not depend on a symbolic variable; in addition, control tokens must be present at the moment that the actual phase is determined.

Fig. 6 shows downsampling by a variable factor n, as represented in the CDDF model. The first time the actor fires, one token is consumed from the control input. The value of this variable (n), expressed symbolically, corresponds to the length of the sequence. Thus, for the next $n-1$ firings of the actor, no tokens are consumed from the control terminal. For other terminals, the token transfer

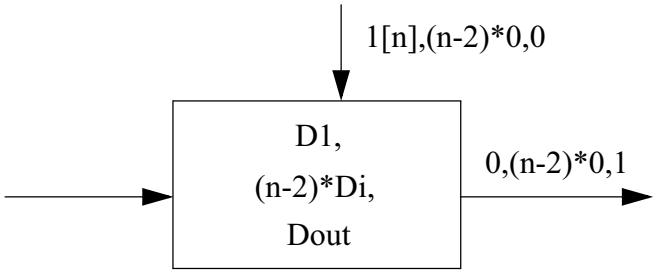


Figure 6. Downsampling by a variable factor in Cyclo-Dynamic Dataflow.

sequence is a function of the value of the first token read from the control edge.

Compared to BDF, CDDF provides a way to express the programmer's knowledge about the internals of an actor, which a tool has no way of knowing otherwise, without attempting to parse the individual library specifications. Hence, some CDDF specifications can be successfully checked for *consistency* (the number of tokens produced on an edge is the same as the number of tokens consumed during one complete cycle), while the corresponding BDF graph cannot be so checked [35]. Thus CDDF specifications possess better analyzability than functionally equivalent BDF specifications.

1.3.5 Heterochronous Dataflow

In ***charts** [18], Girault et al. propose the concept of hierarchical finite state machines with multiple concurrency models, where unlike other concurrent, hierarchical FSM models (like Statecharts [19]), the idea is to decouple the concurrency model from the hierarchical FSM semantics. This allows hierarchical FSMs to be embedded in a variety of concurrency models, including *dataflow*, *discrete event*, and *synchronous/reactive*.

In the dataflow context, the model allows an SDF graph to refine a state of an FSM, and conversely an SDF actor can be refined by an FSM. Interesting possibilities arise when an FSM system has more than one state refined to an SDF graph, and the *type signature*s (the number of tokens produced and consumed on each firing) of the SDF graphs are different. In that case the FSM subsystem cannot be embedded within an SDF graph. This property is exploited to propose a new model

of computation called **Heterochronous Dataflow** (**HDF**), where an actor can have a finite number of type signatures associated with it. When such an actor fires, a well-defined type signature is in effect. But type signatures are allowed to change between firings. This is somewhat similar to CSDF, but the order in which type signatures are used is not cyclic, nor even predictable. When an HDF system starts execution, the initial type signature in effect for every actor is used to solve the balance equations, and find an *iteration*. The semantics demand that each type signature must remain constant for the duration of the corresponding iteration. After the completion of the iteration, a new set of type signatures take effect, and the balance equations must be solved anew to redefine an iteration. Although the number of type signature combinations can be exponential in the number of actors, it is finite. For each combination, all key issues are decidable (deadlock, bounded memory), and schedules can be statically constructed. Thus the HDF model combines SDF and FSM semantics to obtain greater expressivity than SDF alone, but retains the key advantage of SDF (decidability).

1.3.6 Bounded Dynamic Dataflow

Pankert et al. [28] also make use of FSM semantics in introducing control flow into the SDF framework. They propose a heterogeneous model built on top of an extension of SDF, called Scalable Synchronous dataflow (SSDF) [33]. An SSDF actor has the capacity to process any integer multiple of the basic token production/consumption quantities at an output/input port by providing a *block processing* parameter. This can substantially reduce the inter-actor context-switching in synthe-

sized implementations.

SSDF is extended to **Bounded Dynamic Dataflow** (**BDDF**), in which a distinction is made between control flow, and dynamic data flow. The latter is handled by introducing dynamic ports, distinct from normal SSDF ports, where an upper bound is provided for the data rate at each dynamic port to keep the model bounded. The basic strategy for software synthesis is to identify synchronous regions, that include all SSDF blocks, and provide an internal static schedule for each such region. Then a program is generated to handle the dynamic behavior at run-time. Control flow is handled by FSMs. An FSM state consists of an arbitrary set of connected blocks, and each block may be part of multiple states. The set of blocks associated with a state form a subgraph to be activated while the corresponding state is running. A set of transitions is specified for each state. The logical expression of events needed to trigger a transition is evaluated at run-time, and when it becomes true, the successor state is started. Code generation for control flow results in providing a *switch-case* statement, where each state is repeatedly executed until one or a combination of multiple events causes it to be stopped, and another state to be entered.

Chapter 2. Parameterized Dataflow Modeling

We introduce a parameterized dataflow modeling framework that imposes a hierarchy discipline on an underlying dataflow model and allows a subsystem's behavior to be controlled by a set of parameters. These parameters can vary at run-time, thus allowing the subsystem behavior to change dynamically. Parameters can control the functional behavior as well as the token flow behavior of a dataflow graph. The parameter coordination mechanism is such that every parameterized graph always behaves like a graph in the underlying ("unparameterized") dataflow model during each of its invocations, but can assume different configurations across invocations.

Parameterized dataflow modeling differs from the dataflow modeling techniques discussed in Section 1.3 in that it is a *meta-modeling* technique: parameterized dataflow modeling requires only that the underlying model be a dataflow model that has a notion of a graph iteration (invocation), and hence our dataflow parametrization concepts can be incorporated into any dataflow model that satisfies this requirement, to further increase its expressive power. For example, a minimal periodic schedule is a natural notion of an iteration in SDF, SSDF, CSDF, and MD-SDF. Similarly, in BDF, a complete cycle, when it is exists, can be used to specify an iteration of a subsystem. Thus it is possible to combine our parameterization techniques with existing dataflow models like SDF, CSDF, SSDF and MD-SDF to yield more expressivity. Parameterization as a meta-modeling technique is further discussed in

Section 6.2.

For clarity and uniformity, and because SDF is currently the most popular dataflow model for DSP, we develop parameterized dataflow formally in the context of SDF. We present comprehensive development of a *parameterized synchronous dataflow* (PSDF) model, which introduces increased dynamic behavior into the SDF framework in a controlled fashion.

Girault's work on *charts with hierarchical FSMs and multiple concurrency models is another example of meta-modeling. Bounded dynamic dataflow also seeks to combine different models of computation — dataflow and finite state machines. Parameterized dataflow differs from these in that it does not require any departure from the dataflow framework. This may be advantageous for users of DSP design tools who are accustomed to working purely in the dataflow domain. However, in a broader sense, our parameterized dataflow approach is in the same spirit as Girault's *charts: our goal is to enhance the effectiveness of a range of existing and conceivable models of computation rather than dogmatically advocating a single new technique.

2.1. Parameterized Synchronous Dataflow

PSDF uses a hierarchical parametrized representation, where hierarchy is used to denote control dependency, and parameters (expressed as symbolic variables) are used to control the functional and dataflow behavior of hierarchical subsystems. For every subsystem, associated parameters maintain a constant value during one invocation of that subsystem, thus constraining the PSDF subsystem to

behave as an SDF subsystem on every invocation. Dynamism is introduced by allowing the parameters to assume different values across invocations of a subsystem. This gives rise to a "locally synchronous, globally dynamic" property. We further elaborate on this concept in Section 2.2.

A *PSDF graph* is a dataflow graph consisting of *PSDF actors* and *PSDF edges*. A PSDF actor is characterized by certain parameters, where both the actor's functionality as well as its dataflow behavior (number of tokens consumed/produced at different ports — where an edge is incident on it — of an actor) can depend on these parameters.

For example, a PSDF *downsampler* actor can be characterized by two parameters {*factor*, *phase*}, where *factor* is the decimation ratio [34], and *phase* denotes the index of the input token that is actually transferred to the output. The functionality of the downsampler actor depends on both these parameters, while the dataflow behavior depends only on the *factor* parameter. More precisely, the number of tokens consumed by the actor depends on *factor*, and the number of tokens produced by the actor is fixed at one, being independent of any of its parameters. In addition to these externally-visible parameters (*external parameters*), a PSDF actor can have some *internal parameters* (like state information) that are not visible outside. Henceforth, when we say PSDF actor parameters, we refer to the external parameters.

From the example of the downsampler actor, we see that among the external parameters, we can distinguish between the *external-dataflow parameters* (e.g., *fac-*

tor), and the *external non-dataflow parameters* (e.g., *phase*). The dataflow behavior of an actor (token production and consumption) depends on its external-dataflow parameters, but does not depend on its external-nondataflow parameters.

In this context, it is worthwhile to take a closer look at the different roles played by parameters and dataflow inputs in a PSDF actor. Every time an actor is invoked, it consumes some tokens from its dataflow inputs, and can perform certain actions depending on the value of the consumed tokens. The value of a dataflow input can change across every invocation of the associated actor, and hence can be used to control the behavior of the actor at the granularity of every actor invocation. On the other hand, a parameter, in general, is set ("produced") once and maintains a constant value for a sequence of successive invocations of the actor. Thus parameters control actor behavior at a coarser level of granularity than dataflow inputs.

In the simplest case, parameters are assigned static values that are maintained throughout all invocations of the entire application. The PSDF model allows more fine-grained control over the time period (in units of number of invocations of the enclosing subsystem) throughout which parameters maintain constant values. As an aside, note that although external actor parameters usually maintain constant values across multiple invocations of an actor, internal actor parameters (like the state information in a FIR filter) can change on every invocation. We will derive similar properties of subsystem parameters shortly.

A PSDF edge has three characteristics:

- the number of units of delay on the edge (the number of initial tokens);

• the value of the initial token (called the *delay value*) associated with each unit of delay;

• the number of invocations of the sink actor after which the associated delay values are to be re-initialized (*re-initialization period*);

Like a PSDF actor, a PSDF edge can also have parameters that determine its characteristics.

An application is modeled in PSDF by a *PSDF specification*, also called a *PSDF subsystem*. Usually, an application is naturally expressible in terms of certain parameters. For example, in a block adaptive filtering application, the size of each block, and the filter order are two natural parameters of the application [21]. Thus, PSDF subsystems also have an intuitive concept of parameters, similar to PSDF actors. Some of the parameters of a PSDF subsystem may be visible externally in the enclosing graph (*external subsystem parameters*), while some of the parameters may not be visible externally (*internal subsystem parameters*). These parameters together comprise the *immediate parameter* set of a subsystem. When we say parameters of a subsystem, we usually refer to these immediate parameters, unless otherwise stated.

The external subsystem parameters can again be divided into *external-dataflow*, and *external-nondataflow* parameters, depending on whether or not they affect the dataflow behavior of the subsystem. Thus, from an enclosing hierarchical subsystem, a PSDF subsystem appears just like a PSDF actor. Indeed, the PSDF model naturally supports a hierarchical specification format, where a PSDF graph can con-

tain any number of child PSDF subsystems that are abstracted as PSDF actors. This semantic hierarchy, called *control hierarchy*, is distinct from syntactic hierarchy, which is allowed and supported in the usual fashion (the system is equivalent to flattening the syntactic hierarchy).

Every PSDF subsystem is divided into a (possibly empty) control flow part, a data flow part, and a (also possibly empty) mixture of control flow and data flow. These appear in the form of three PSDF graphs called the *init* graph, the *body* graph, and the *subinit* graph. The body graph models the dataflow functionality of the subsystem, while the init and subinit graphs are used to control the behavior of the body graph by appropriately configuring subsystem parameter values in actors present in the init and subinit graphs.

The init graph models pure control flow in the sense that it neither accepts any dataflow inputs from outside, nor produces any dataflow outputs, and it is responsible for configuring the external-dataflow subsystem parameters. The subinit graph accepts dataflow inputs, but does not produce any dataflow outputs, and it configures all the internal subsystem parameters. Conceivably, the subinit graph can use information available at its dataflow inputs in configuring the value of an internal subsystem parameter. The reason why the init graph does not accept dataflow inputs for similar purposes lies in the invocation semantics of PSDF. The subinit graph is invoked as an inherent part of the dataflow specification of the parent graph in which the subsystem is embedded as an hierarchical actor, while the init graph is invoked in a disjoint fashion, separately, at the beginning of every invocation of the

parent graph. Thus, it is natural for the subinit graph, but not for the init graph, to accept dataflow inputs from parent graph actors. The purpose of the init and subinit graphs is to configure parameters, and hence none produces any dataflow outputs.

The external-nondataflow subsystem parameters can be configured either by the init graph, or by any actor in the subsystem's parent graph that is a (dataflow) predecessor of the subsystem. In the latter case, there is a dataflow edge from the parent graph actor to the subsystem, and the corresponding parameter is bound to this dataflow edge.

We have seen that the external parameters can be divided into external-dataflow and external-nondataflow parameters, depending on how they affect the dataflow behavior of the subsystem. On the basis of how the external parameters are configured, they can also be divided into two groups: *init-configured external parameters* (or simply *init-configured parameters*) that are configured by the init graph, and *parent-configured external parameters* (or simply *parent-configured parameters*) that are configured by parent graph actors. From this viewpoint, internal subsystem parameters are also referred to as *subinit-configured parameters*. The subinit-configured parameters and the parent-configured parameters are together referred to as *non-init-configured parameters*. The init-configured external parameters include all the external-dataflow parameters, while the parent-configured external parameters is a subset of the external-nondataflow parameters. Every subsystem also inherits the parameter set of its parent subsystem (*inherited parameter set*).

Within a PSDF specification, actor parameters in a PSDF graph can be

assigned fixed static values, or they can be assigned subsystem parameter values. In the latter case, we say that the PSDF graph *uses* those subsystem parameters. Parameter values are set and used in a *producer-consumer* fashion on a graph by graph basis, and thus, a PSDF graph does not use parameters that it sets. In a subsystem specification, the purpose of the init and subinit graphs is to completely specify the behavior of the body graph, either by computing values for each parameter that the body graph uses, or by "passing on" the value from an inherited subsystem parameter value. Thus, it is not necessary for the body graph to use any inherited parameters, it only uses init-configured external parameters, and internal parameters of the subsystem that it belongs to. In configuring the behavior of the body graph, the init and subinit graphs utilize information set up in the init and subinit graphs of hierarchically higher-level subsystems by using inherited parameters of the associated subsystem. In addition, the subinit graph also makes use of the (init-configured and parent-configured) external subsystem parameters values. The usage pattern mentioned here is sometimes referred to as the PSDF *scoping rules*. The motivation for this usage pattern will become more clear when we introduce the local synchrony concepts in Section 2.2, and during the formal development of the PSDF model in Chapter 3.

The invocation semantics of a subsystem H is as follows. The init graph of a subsystem H is invoked once at the beginning of each invocation of the subsystem's parent graph, G. Furthermore, each actor in the init graph that configures an external parameter value is constrained to fire only once per invocation of the init graph.

One invocation of H comprises one invocation of the subinit graph, followed by one invocation of the body graph. If there are any parent-configured external parameters that are bound to dataflow input edges of H, then H consumes one token from each of those edges on every invocation. In the subinit graph also, actors responsible for configuring internal parameter values are constrained to fire once per invocation of the subinit graph. Thus, in this scenario, the subsystem internal parameters and the parent-configured external parameters maintain constant values over one invocation of the subsystem, but can change values across invocations, while the init-configured external parameters maintain constant values over one invocation of the parent graph. Note that this is conceptually similar to what happens in a PSDF actor.

From the perspective of invoking a PSDF subsystem, the application designer can assign values to actor parameters in the body graph of a subsystem in one of three ways:

- if the parameter is intended to maintain a constant value over the entire duration of the application, then assign a static fixed value;
- if the parameter is intended to maintain a constant value over every invocation of the enclosing parent graph of the subsystem, then assign those subsystem parameter values that are set up in the init graph (init-configured external parameters);
- if the parameter is intended to maintain a constant value only over one invocation of the subsystem, but in general, assume different values across subsystem invocations, then assign those subsystem parameter values that are set up in

the subinit graph (internal subsystem parameters);

Similarly, actor parameters in the subinit graph that are intended to change across every invocation of the subsystem are assigned parent-configured external subsystem parameter values, while init-configured external parameters or inherited parameters are used depending on whether those actor parameters will maintain constant values over every invocation of the parent graph, or over every invocation of some ancestor graph further higher up in the hierarchy tree.

As a part of a PSDF specification, the token flow at a port of a PSDF actor can be specified by the programmer as a statically fixed integer, as a symbolic expression of the actor's parameter set, or as an *unspecified value*. In the third case, the programmer has to provide a software subroutine, called the *parameter interpretation function* of the actor, which will take as input an actor port, and an assignment of values to its parameter set, and will produce as output the token flow at the specified port for that particular configuration of its parameter set. Similarly there is a notion of a parameter interpretation function of an edge that the programmer can optionally provide, if the associated characteristics of that edge have not been statically or symbolically specified.

2.2. Local Synchrony

As mentioned before, the init graph of a subsystem does not participate in any dataflow outside the graph. The subinit graph can accept dataflow inputs from outside, but does not produce any dataflow outputs. The body graph of course, both accepts and produces dataflow at its interface ports. Thus the token flow at the ports

of a subsystem can potentially depend both on the internal parameters as well as on the external parameters of the subsystem. However, the locally synchronous semantics of PSDF forbids subsystem token flow to depend on internal subsystem parameters, or on parent-configured external subsystem parameters, the latter being a subset of the external non-dataflow subsystem parameters. The reason for this restriction can be explained from two different perspectives — philosophically, and from a scheduling viewpoint.

Recall, that the token flow at an actor port is a function of the external-dataflow parameter set of the actor, and does not depend on the actor's external-nondataflow parameters or internal parameters. In an analogous fashion, from a philosophical perspective, subsystem token flow should also depend only on the external-dataflow subsystem parameters, and not on the internal subsystem parameters, or external-nondataflow subsystem parameters. Thus, the functional behavior of the subsystem should naturally lead to parameter choices such that the interface token flow behavior of the subsystem satisfies the local synchrony condition.

From the semantic perspective, local synchrony is necessary for schedulability of a PSDF graph. Along with the parameter configuration mechanism and the invocation semantics, local synchrony of a subsystem ensures that each PSDF graph in a specification always behaves as an SDF graph in each of its invocations. For a subsystem embedded in a PSDF graph G, the init graph of the subsystem is invoked at the beginning of an invocation of G. Since, the init graph uses only inherited parameter values, which have already been fixed up by the init and subinit graphs of

hierarchically higher-level subsystems, it behaves as an SDF graph in each of its invocations. The init graph configures the external-dataflow subsystem parameters to fixed values at the beginning of each invocation of G, so the token flow of every hierarchical actor in G is fixed, and the parent graph G behaves as an SDF graph during each invocation. Similarly the external-nondataflow parameters of H are set to fixed values either by the init graph or by actors in G before H is fired. So, in each of its invocation, the subinit graph of H uses only fixed parameter values, and thus behaves as an SDF graph. Again, an invocation of the subinit graph of H fixes up the internal parameters of H, so that the body graph of H also uses only fixed parameter values in each of its invocation, and is equivalent to an SDF graph in that period.

2.3. PSDF and Other Dataflow Models

The parameterization concept appears naturally in an application modeling context, and along with the underlying SDF model, it makes the powerful semantics of PSDF intuitive and easy to understand. Fig. 7 shows an example of the PSDF representation of the downsampler actor. Compared to the CDDF representation of Fig. 6, the PSDF version is clearly a more concise representation.

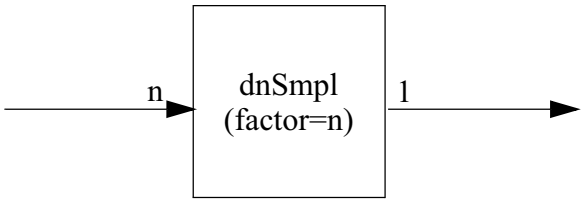


Figure 7. Downsampling by a variable factor in PSDF.

Some of the concepts in PSDF have been explored before in other models in different contexts or with different treatments. As discussed in Section 1.3, the BDF and CDDF model also make use of symbolic variables in some form. The MD-SDF model addresses the issue of re-initialization of delays. In fact, Lee has also explored the re-initialization concept in the context of the SDF model [25]. HDF has the concept of a leaf SDF actor having different token consumption-production type signatures, although these have to be statically specified. The idea of actor parameters is well known in block diagram DSP programming environments. Conventionally, these parameters are assigned static values that remain unchanged throughout execution.

Our parameterized dataflow approach takes the concept of actor parameters as a starting point, and develops a comprehensive framework for dynamically reconfiguring the behavior of dataflow actors, edges, graphs, and subsystems. In addition, the parameterized framework provides a systematic formalism that unifies various ideas that have been explored before in a disjoint fashion into one powerful unit. For example, SSDF has the concept of *vectorization* of a SDF graph, by providing loops around an SDF actor with an optional block processing factor. In PSDF, this block processing factor can be modeled as an actor parameter that can be statically configured, dynamically re-configured, or both. Dynamic re-configuration of the block processing factor may be desirable to adapt an application's performance to time-varying application constraints involving metrics such as latency, throughput, and power consumption.

2.4. Examples of PSDF Specifications

In this section, we illustrate the syntax and semantics used to model an application in the PSDF representation, through four examples. The first example shows a portion of a generic PSDF specification that exercises all the features of the PSDF model. The next three examples demonstrate concrete applications. PSDF scheduling of Examples 2, 3, and 4 will be illustrated in Section 4.3.6.

We use the following notation. Hierarchy is denoted by enclosing a block in double rectangles. The immediate parameter set of a subsystem is specified inside the subsystem with the notation

$$params = \{<param1>, <param2>, ...\}.$$

Inside an actor, actor parameters are represented in the left hand side of an equation, and are assigned suitable values on the right hand side. For brevity, we have sometimes only included those actor parameters that determine the dataflow behavior of that actor, whenever that does not affect the macro-level functionality of the application. The notation (*sets* <*param1*>, <*param2*>, ...) inside an actor, is used to denote the configuration of those parameter by that actor. If an actor port is not marked with token flow information (either a static integer, or a symbolic expression of its parameters), that implies that the programmer has left it unspecified, and has provided a suitable parameter interpretation function. A delay on an edge is denoted by placing a triangular mark on that edge. For n $(n > 1)$ units of delay, the edge is labeled with nD, followed by comma separated expressions in parentheses that provide the initial values and re-initialization periods of the tokens. Each expression

consists of two parts, which are again separated by a comma. The first part specifies the initial value, while the second part specifies the re-initialization period (0 denotes that re-initialization is not necessary). For n units of delay, if less than n expressions are provided, the remaining delay tokens take on properties from the last expression in the sequence.

Example 1: *Fragment of a generic PSDF specification*

Fig. 8 shows a component of a generic PSDF specification. Actor parameters are denoted by the prefix *Ap*, and subsystem parameters start with the prefix *Sp*. The graph G consists of four non-hierarchical actors and one hierarchical actor H. The subsystem corresponding to H has three subsystem parameters — *Sp2*, *Sp3*, and *Sp4*, which comprise the immediate parameter set of H. The init graph of H has a single actor that uses the inherited subsystem parameter *Sp1*. Among the immediate parameters, *Sp2* is an external-dataflow parameter that is set by the init graph (init-configured external parameter); *Sp3* is an external-nondataflow parameter that is set by the parent graph actor $G3$ (parent-configured external parameter); and *Sp4* is an internal parameter that is set by the subinit graph actor *S2*. Different actor parameters have been assigned different subsystem parameter values, in accordance with all the scoping rules. Except for the output port of body graph actor *B2*, the token flow at all other ports of all other actors have been statically specified, either as fixed integers, or as symbolic expressions of the subsystem parameters on which the token flow depends. For example, the body graph actor *B1* uses the subsystem parameter *Sp4*, and consumes a fixed quantity of two tokens on its input port, and produces

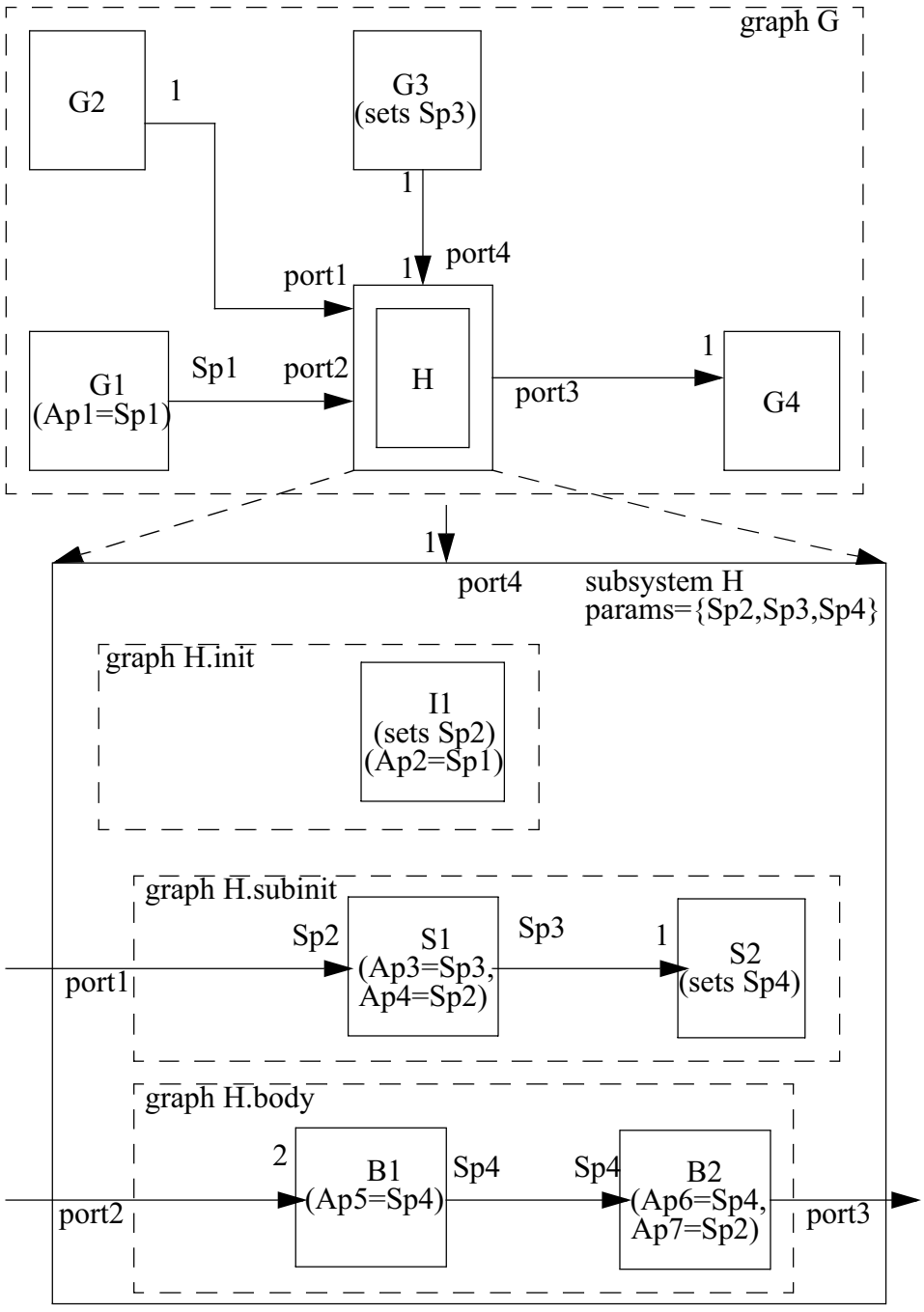


Figure 8. A component of a generic PSDF specification.

Sp4 number of tokens on its output port. For actor *B2*, the user has provided a suitable parameter interpretation function to determine the token flow at its output port at run time. Let us denote this unknown token flow by the symbolic variable *UB2O*.

For local synchrony verification of subsystem H, the token flow at all the dataflow interface ports of the subinit graph and the body graph of the subsystem have to be computed and checked for proper containment. The token flow at the input port of the subinit graph (*port1*) is given by *Sp2*, which is a function of an init-configured parameter, while the token flow at the input port of the body graph (*port2*) is statically fixed at 2. However, the token flow at the output port of the body graph (*port3*) is equal to the unknown quantity *UB2O*, and hence cannot be determined at compile-time. From the parameter configuration of actor *B2*, it can be seen that *UB2O* can in general, depend on the subsystem parameters *Sp2* (external-dataflow parameter) or *Sp4* (internal parameter). Recall that for the subsystem to be locally synchronous, *UB2O* must be independent of *Sp4*, since *Sp4* is configured in the subinit graph and not in the init graph. Since actor parameter *Ap6* is assigned the subsystem parameter *Sp4*, local synchrony of the subsystem will depend on whether or not *Ap6* is an external-nondataflow parameter, or an external-dataflow parameter of the actor, which will determine the dependence of *UB2O* on *Sp4*. Note that *port4* is not a dataflow interface port for either the subinit graph, or the body graph. The subsystem parameter *Sp3* is bound to this port, and the subsystem always consumes a single token from this input port on each invocation. Thus, token flow at *port4* is known at compile-time, and is not relevant for local synchrony verification.

Example 2: *Computation of Fibonacci Numbers, based on user input*

In this example (Fig. 9), the PSDF subsystem *fib* has a single parameter, denoted by the symbolic variable p, which corresponds to the Fibonacci Number being computed. In the init graph of the subsystem, the *setFib* actor sets the value of p (e.g., from user input, or by random number generation). In the dataflow part of the subsystem, the body graph uses the parameter p. The *add* actor has both of its inputs coming as feedback edges from its output. The first input has one unit of delay initialized to one. The second input has two units of delay, both initialized to zero. Both the delay values have to be re-initialized after p invocations of the *add*

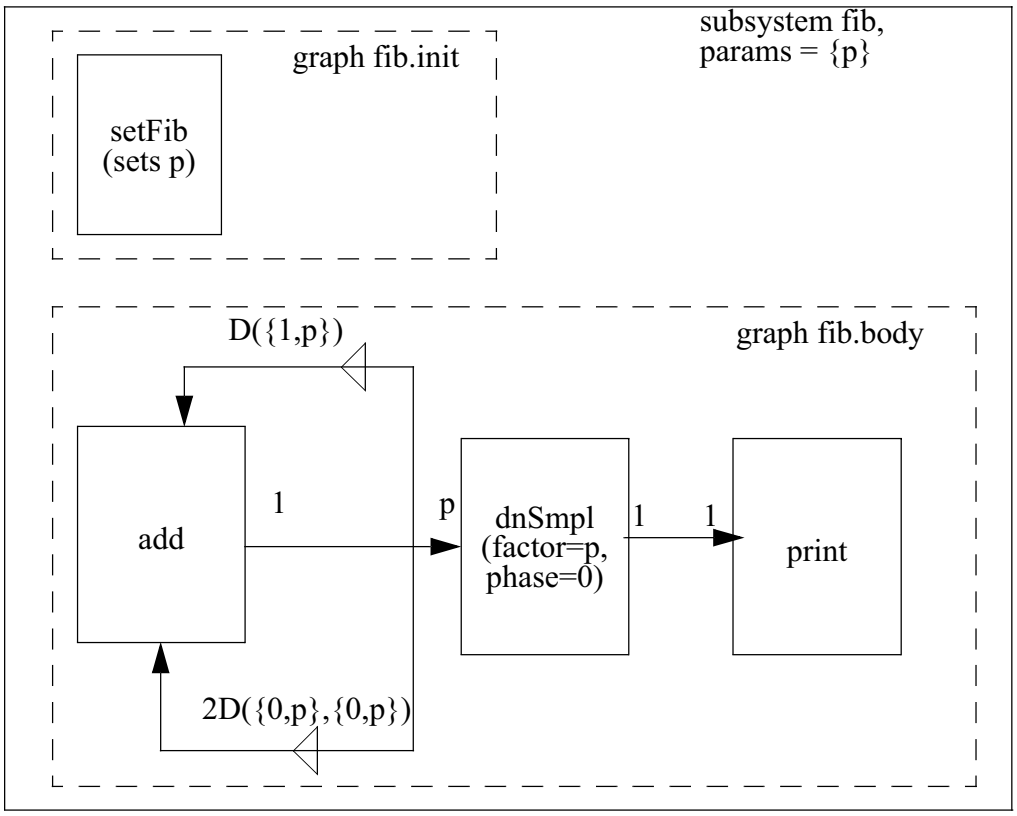


Figure 9. Computation of the pth Fibonacci Number in PSDF.

actor. The output of this addition goes to the *dnSmpl* actor, which downsamples by a factor of p . The actor parameter *factor* of the *dnSmpl* actor is set to p , while the *phase* parameter is set to 0, such that the most recent input sample is transferred to the output. The downsampling result is passed on to the *print* actor which prints the result.

Fig. 10 shows the corresponding SDF system for computing the seventh Fibonacci Number. As is evident from the two representations, the PSDF representation is a straightforward and intuitive extension of the underlying SDF model to compute an arbitrary sequence of Fibonacci Numbers.

Example 3: *Weighted average of variable length data packets*

In this example (Fig. 11), the objective is to compute the weighted average from an input data stream produced by the *genData* actor. The data stream has to be broken up first into packets, and then into frames. Each frame consists of one or more contiguous packets, and each packet consists of one or more contiguous

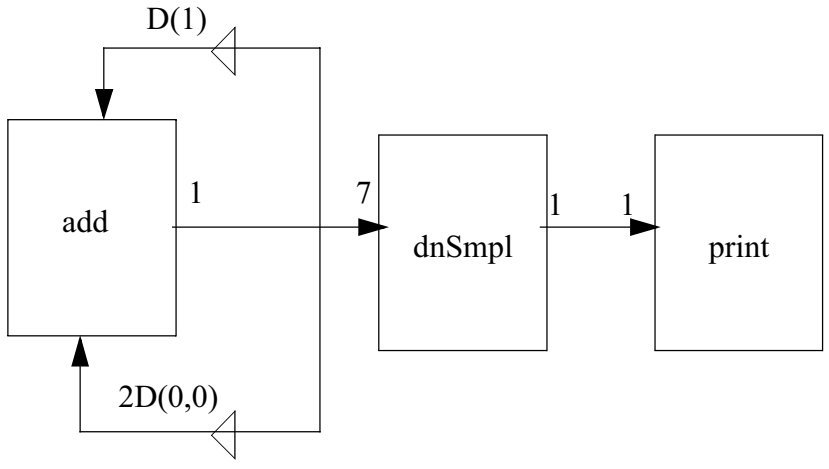


Figure 10. Computation of the 7th Fibonacci Number in SDF.

tokens. The length of each packet, and the number of packets in a frame are specified at run-time by the *genHdr* SDF actor. The weighted average of each frame has to be computed, with respect to the weighted value of each packet. The corresponding PSDF subsystem *wtAvg* is modeled as having two parameters, denoted by the symbolic variables *plen*, and *flen*. The length of each packet is denoted by *plen*, and *flen* gives the length of each frame. In the init graph of the subsystem, the *readHdr* actor reads these from the two tokens produced by the *genHdr* actor, and correspondingly sets *flen* and *plen*. To compute the weighted average, in the body graph of the subsystem, the data values in each data packet (of length *plen*) are multiplied

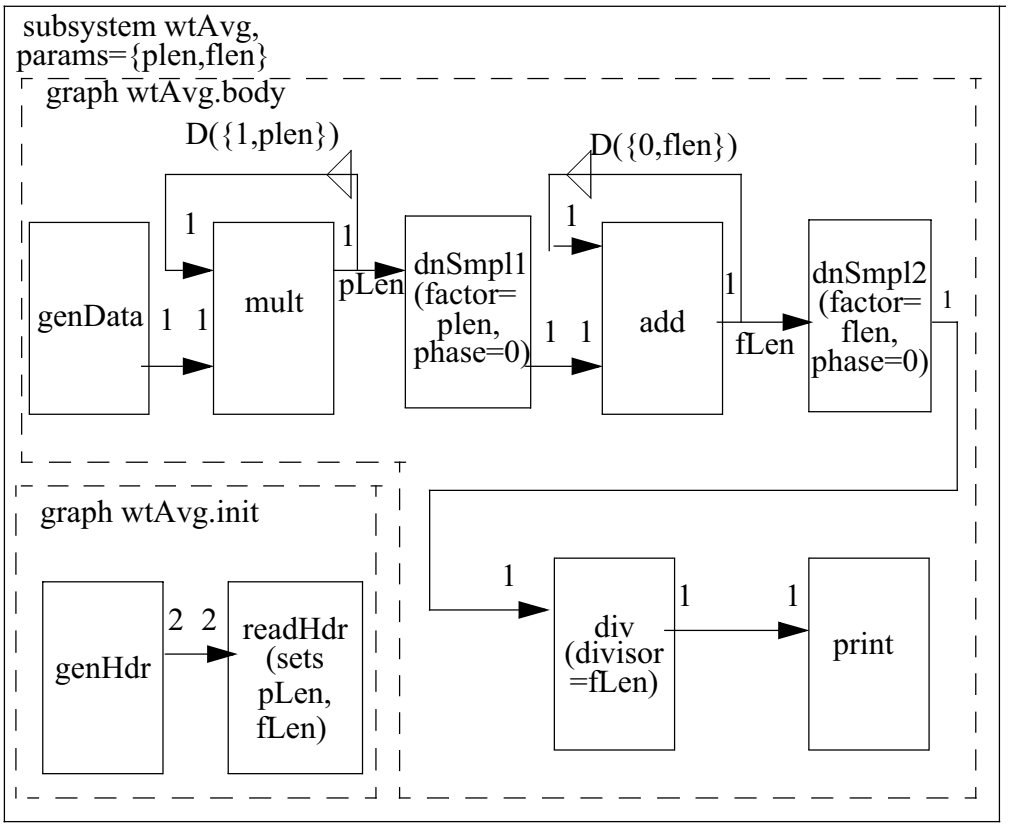


Figure 11. Weighted average of variable length data packets in PSDF.

together by the *mult* actor, then the product for each packet is added up by the *add* actor (for a total of *flen* additions), and finally this sum is divided by *flen* in the *div* actor, to obtain the weighted average. Here, *div* is a PSDF actor, where the divisor is specified as an actor parameter (*divisor*). In this example, the actor parameter *divisor* is assigned the value of the subsystem parameter *flen*. Finally, the result of the computation is printed by the *print* actor. Re-initialization of the corresponding delay values are necessary for every *plen* invocations of the *mult* actor, and for every *flen* invocations of the *add* actor.

Example 4: *Prediction Error Filter*

This example (Fig. 12) models a prediction error filter, where a linear adaptive filter implementing the least-mean-square (LMS) algorithm [21] is used to provide the best prediction of the present value of a random signal. The present value of the signal provides the desired response of the adaptive filter, while past values of the signal supply the input applied to the adaptive filter, and the estimation (prediction) error serves as the output.

In the PSDF model of the application, shown in Fig. 12, the topmost subsystem *predictor* has empty init and subinit graphs. The body graph makes use of several instances of the *fork* actor. The functionality of the *fork* actor is to simply replicate its input to each of its outputs. The outputs of the first fork actor (*fork1*) provide the current value of the random signal. One output serves as the desired response of the adaptive filter (*port3*), while the other output is delayed and supplied to the second fork actor (*fork2*), which provides the past value of the random signal

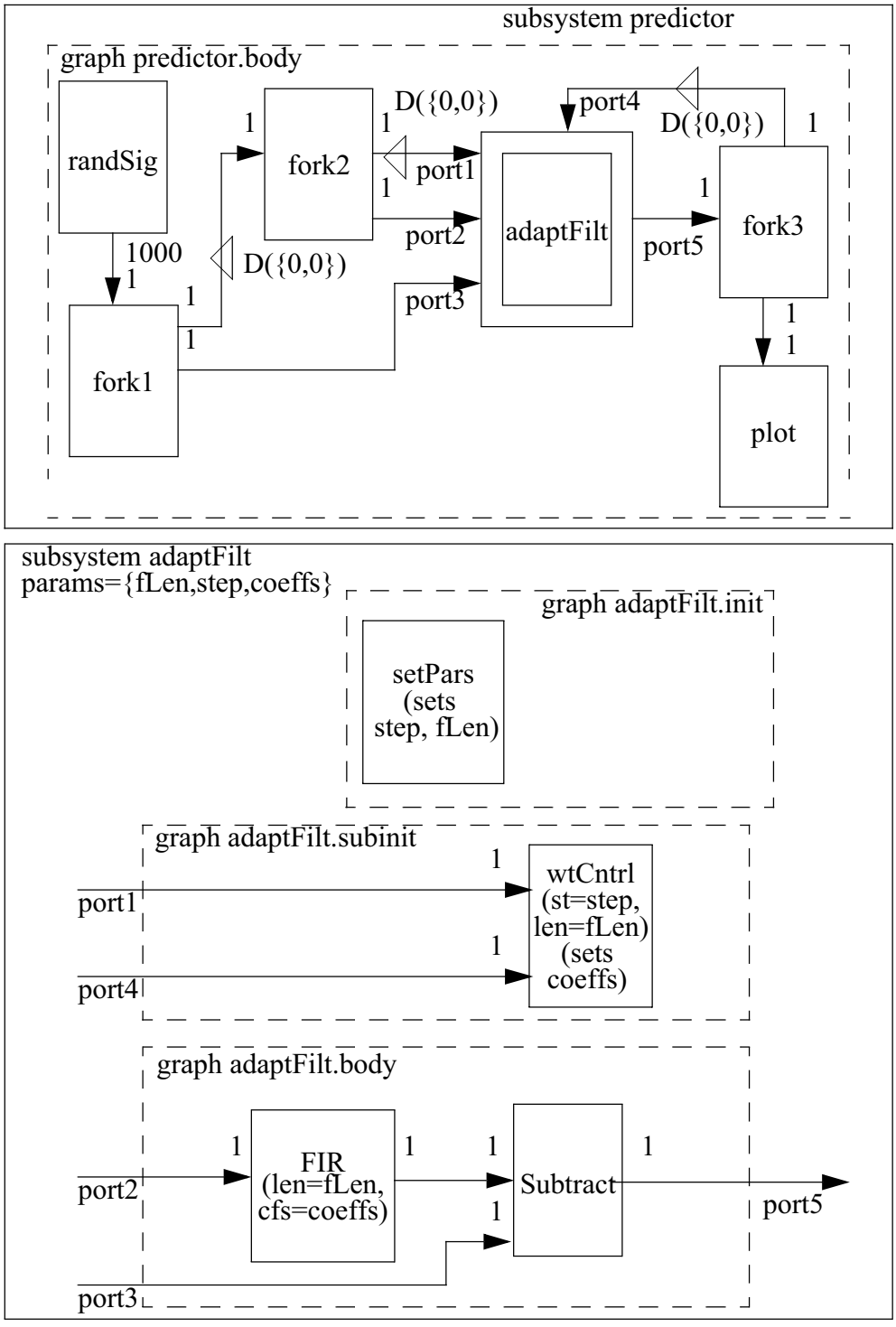


Figure 12. A prediction error filter in the PSDF model.

at its two outputs. One output of *fork2* goes into the adaptive filter as the input signal (*port2*), while the other output is again delayed and provided as input to the weight control mechanism of the adaptive filter (*port1*). The (error) output of the adaptive filter is plotted by the *plot* actor and supplied to the weight control mechanism after a delay (*port4*).

Control hierarchy is used to model the LMS adaptive filtering process via subsystem *adaptFilt*. The parameters of *adaptFilt — fLen*, *step*, *coeffs*, model the adaptive FIR filter length, the step size of the adaptation weight control mechanism, and the coefficients of the FIR filter, respectively. In the init graph of the subsystem, appropriate values are assigned to the step size and the filter length in the *setPars* actor, while in the subinit graph, the *wtCntrl* actor accepts the prediction error and current value of the random signal as inputs, and uses the step size and filter length parameters to update the filter coefficients of the *FIR* actor for filtering the next input sample. At the first invocation of the *wtCntrl* actor, both its inputs have delay tokens initialized to 0, and the actor provides an initial guess of the filter weights. Actor parameters of the *FIR* and *wtCntrl* actors are assigned suitable subsystem parameter values, as shown in the figure. The *Subtract* actor determines the prediction error from the filter output and the desired response.

In this example, a fixed step size and filter length are used for a thousand input samples. In multiple runs of the application, the programmer can experiment with different values of the step size and filter length, to minimize the prediction error. Depending on application-specific requirements, the programmer can also

control the length of the input samples over which a fixed step size and filter length are used by making that a parameter of the topmost subsystem *predictor*, and configuring the parameter in the *predictor* subsystem's init graph.

In the *adaptFilt* subsystem, the step size (*step*) and the filter length (*fLen*) can be considered as external subsystem parameters (external-nondataflow in this case) that are visible outside, and serve as handles to control the subsystem functionality. These remain fixed over one invocation of the subsystem's parent graph. On the other hand the *coeffs* parameter, modeling the FIR filter coefficients, is an internal subsystem parameter that the subsystem uses for its local purposes, and which may change value across every invocation of the subsystem.

In this example, one invocation of the *adaptFilt* subsystem comprises single invocations of the *wtCntrl*, *FIR*, and *Subtract* actors. Since, the *coeffs* internal parameter changes across every invocation of the subsystem, the *FIR* actor effectively receives a new set of coefficients on each invocation. Recall our discussion on dataflow inputs and parameters of an actor (Section 2.1), and observe that the filter coefficients of our prediction error filter example can also be modeled as a dataflow input of the actor. This alternative model of the *adaptFilt* subsystem is shown in Fig. 13. In this model, *coeffs* is no longer an internal parameter of the subsystem; instead, the *wtCntrl* actor supplies *fLen* number of filter coefficients to the *FIR* actor as a dataflow input, and the subsystem now has an empty subinit graph. This demonstrates that, analogous to other programming models, there is no one correct way of modeling in PSDF — rather there can be several ways of modeling the same func-

tionality, and which one the programmer chooses in practice may be guided by various application-specific considerations, for example, the version of the *FIR* filter that is already available in the actor library. In Section 3.9, we further elaborate on the relationship between dataflow inputs and actor parameters.

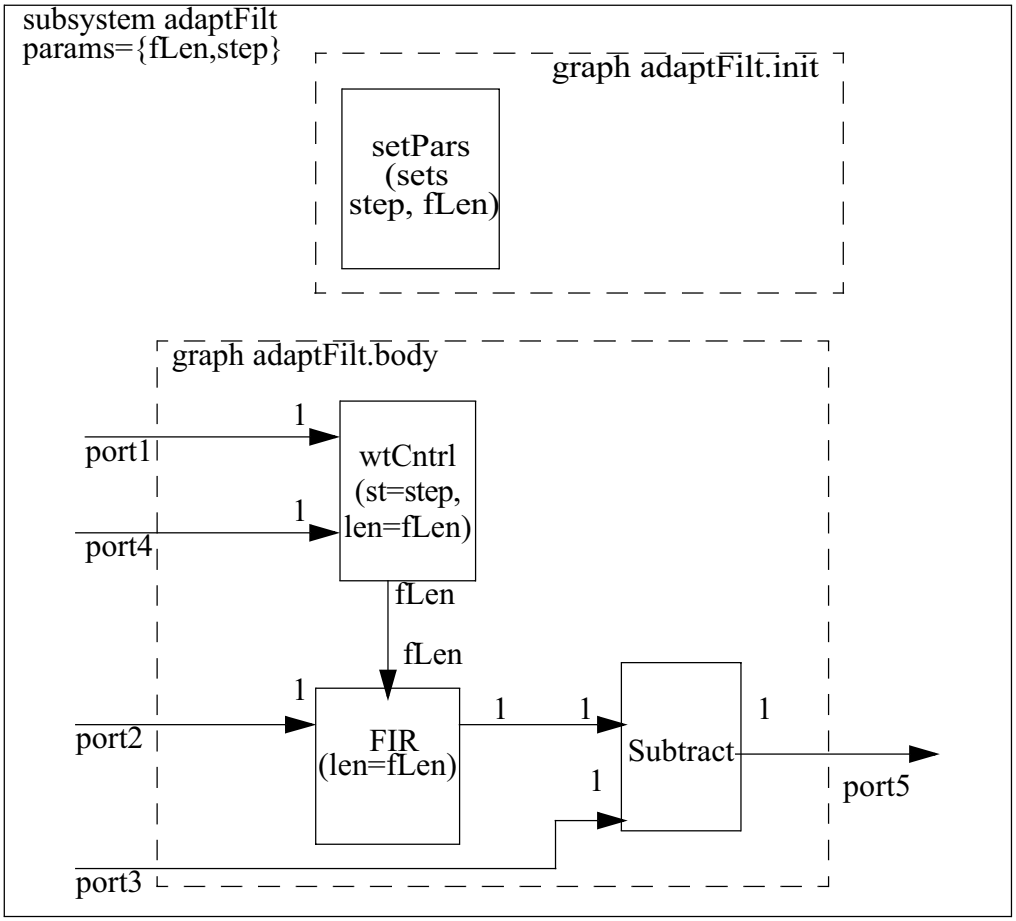


Figure 13. The *adaptFilt* subsystem in an alternative modeling style for the predictor error filter example.

Chapter 3. Formal Semantics of PSDF

The previous chapter has introduced PSDF concepts in an intuitive, informal manner. In this chapter, we introduce some formalization that enables more precise specification and analysis of PSDF semantics.

3.1. Parameters

A parameter set is a finite set $P = \{p_1, p_2, \ldots, p_n\}$ of objects such that each p_i has an associated domain, denoted $domain(p_i)$, which is a finite and nonempty set. Each p_i is called a parameter of P. A special object, which we call the *unspecified parameter value*, and denote by "$\perp$", is reserved for use in incomplete parameter settings. Thus, every parameter set $\{p_1, p_2, \ldots, p_n\}$ must satisfy $\perp \notin domain(p_i)$. A parameter p_i has a *default value* — $default(p_i)$ associated with it such that

$$default(p_i) \in (domain(p_i) \cup \{\perp\}). \tag{3}$$

For example, consider a parameter set $W = \{x, y\}$, with $domain(x) = \{1, 2\}$, and $domain(y) = \{a, b, c\}$. Then the default values of the parameters can be specified as $default(x) = 2$, and $default(y) = \perp$.

Now suppose that $P = \{p_1, p_2, \ldots, p_n\}$ is a given nonempty parameter set. A *configuration* of P is a $|P|$-element subset of ordered pairs

$$\{(p_1, v_1), (p_2, v_2), \ldots, (p_n, v_n)\}$$

such that for $i = 1, 2, \ldots, n$, $v_i \in (domain(p_i) \cup \{\perp\})$. Each v_i is said to be the *value* of parameter p_i under the given configuration. If $C = \{(p_1, v_1), (p_2, v_2), \ldots, (p_n, v_n)\}$ is a configuration of a parameter set (also called a *parameter configuration*), $C(p_i)$ denotes the value v_i for each i. If for each $p \in P$, $C(p) \neq \perp$, we say that the configuration C is *complete*. Otherwise, C is a *incomplete*. An empty parameter set has an empty configuration, and an empty configuration is always complete. For example, for the parameter set W, $\{(x, 2), (y, b)\}$ is a complete configuration, while $\{(x, \perp), (y, a)\}$ is an incomplete configuration.

Each of the sets $domain(p_i)$ can be viewed as the domain of a parameter that contributes to defining the precise configuration of some higher level object. In the context of the PSDF model, these higher level objects include graph actors and edges. However, not all combinations of permissible individual parameter values are necessarily acceptable. The *configuration domain* associated with a nonempty parameter set P, denoted $domain(P)$ [1], is the set of compatible *valid configurations* of P in the context in which P is being used. Thus,

$$domain(P) \subseteq \{\{(p_1, v_1), (p_2, v_2), \ldots, (p_n, v_n)\} \mid v_i \in (domain(p_i) \cup \{\perp\}) \forall i\}, \quad (4)$$

and any configuration $\bar{p} \notin domain(P)$ can be viewed as an error in the configuration of P. For an empty parameter set, an empty domain is associated with it. For

1. We will occasionally "overload" certain symbols, such as *domain*, in cases where there are more than one closely related meanings. In such cases, the meaning will be clear from the arguments of the overloaded symbols, or from context.

example, in the parameter set W, $1 \in domain(x)$, and $a \in domain(y)$, but these two values may not be compatible together, making $\{(x, 1), (y, a)\}$ an invalid configuration for W. This concept of a valid configuration will become more clear in Section 3.2, when we discuss PSDF actor parameters. Henceforth in this thesis, we consider only valid configurations, unless otherwise stated.

The subset of $domain(P)$ that consists of all configurations that are both valid and complete is denoted $\overline{domain(P)}$.

Given a configuration C of a nonempty parameter set P, and a nonempty subset of parameters $P' \subseteq P$, the *projection of C onto P',* denoted $C|P'$, is defined by

$$C|P' = \{(p, C(p)) | (p \in P')\}. \tag{5}$$

Thus, the projection is obtained by "discarding" from C all values associated with parameters outside of P'. Projecting a configuration (empty or nonempty) onto an empty configuration produces the empty configuration. For example, for the parameter set W, given a parameter subset $W' = \{x\}$, and a configuration $C_1 = \{(x, 2), (y, c)\}$, $C_1\big|W' = \{(x, 2)\}$.

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

$$((C_1\big|(P - P')) = (C_2\big|(P - P'))) \Rightarrow (f(C_1) = f(C_2)). \tag{6}$$

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

If $\bar{P}$ is a family of m disjoint, parameter sets $P_1, P_2, \ldots, P_m$ with associated domains $domain(P_1), domain(P_2), \ldots, domain(P_m)$, the *joint domain* of $\bar{P}$, denoted $Jdomain(\bar{P})$ is defined by

$$Jdomain(\bar{P}) = \left\{ \left(\bigcup_{i=1}^{m} C_i \right) \middle| \text{for each } i, C_i \in domain(P_i) \right\}. \quad (7)$$

If a set of parameterized objects with disjoint parameter domains are combined into a single composite object, the resulting composite parameter domain will not necessarily be equal to the joint domain of the two component objects. The actual domain may be a proper subset of the joint domain if additional configuration constraints are imposed by the given application context. Such constraints may preclude the joint use of certain pairs of component configurations. In our development of the PSDF model, such situations do not arise, and thus, composite objects inherit the joint domain of their component parts.

We refer to the process of setting unspecified parameter values in an incomplete configuration as the process of *refining* the configuration. Suppose that P is a nonempty parameter set, P' is a nonempty subset $P' \subseteq P$, $C \in domain(P)$, and $C' \in domain(P')$. Then the *refinement of C with respect to C'* is defined by

$$rfmt(C, C') = (C - \{(p, \perp) | (p \in P'')\}) \cup \{(p, C'(p)) | (p \in P'')\}. \quad (8)$$

where

$$P'' = \{p \in P' | ((C(p) = \perp) \textbf{ and } (C'(p) \neq \perp))\} \quad (9)$$

denotes the set of parameters that are contained in both P and P', are unspecified in C, and are not unspecified in C'. Refining a configuration (empty or nonempty) with respect to an empty configuration, maintains the original configuration. For example, for the parameter set W, given $W' = \{x\}$, $C = \{(x, \perp), (y, c)\}$, and $C' = \{(x, 2)\}$, $rfmt(C, C') = \{(x, 2), (y, c)\}$.

3.2. PSDF Actors

To develop PSDF concepts precisely it is useful to employ a slightly more detailed model of dataflow graph topologies than what has commonly been used for analysis of pure SDF graphs. Rather than representing a dataflow specification as a directed multigraph in which vertices correspond to actors and edges connect pairs of vertices, we represent a PSDF graph as a bipartite directed graph in which edges connect actor output ports to actor input ports. In this representation, a *PSDF actor* A has a finite set of input ports $in(A)$, and a finite set of output ports $out(A)$. Actor input and output ports are *unique* in the sense that for any actor A, $in(A) \cap out(A) = \varnothing$, and for any two distinct actors A and B,

$$(in(A) \cup out(A)) \cap (in(B) \cup out(B)) = \varnothing. \quad (10)$$

If p is an input or output port of actor A, we write $actor(p) = A$.

For a PSDF actor, the number of tokens produced or consumed at each port

is in general dependent on one or more parameter values. The *parameter set* of a PSDF actor A is a parameter set, denoted $params(A)$, together with a parameter domain $domain(A) \equiv domain(params(A))$, which defines the set of valid parameter value combinations for A. By $\overline{domain(A)}$, we denote the set of combinations $\overline{domain(params(A))}$ that are both complete and valid.

A *configuration* of A, denoted by $config_A$, is a valid — but not necessarily complete — configuration of $params(A)$. Any configuration $C \notin domain(A)$ for A can be viewed as a syntax error in the underlying block diagram programming language. If for some $p \in params(A)$, $config_A(p) = \perp$, this means that the corresponding parameter value is not statically specified (specified by the programmer). Its value is to be determined dynamically, during run-time. Furthermore, its value may change across different invocations of the enclosing PSDF subsystem.

Example 5: The concept of parameterized actors has been employed in block diagram DSP programming environments for years. Most medium- or large-grained DSP functions, such as FIR and IIR filters, and FFT computations, are naturally parameterizable. As a simple example, consider the downsampler actor (dnSmpl) in Ptolemy [12], which has two parameters $params(\text{dnSmpl}) = \{\text{factor,phase}\}$ that represent respectively, the decimation ratio [34], and the index of the input token to be actually transmitted to the output. The domains of these two parameters are given by $domain(\text{factor}) = \{1, 2, \ldots, M\}$, and $domain(\text{phase}) = \{0, 1, \ldots, M-1\}$, where M is some pre-specified maximum integer value, which could, for example, be determined by the maximum word length on the host computer. The phase

parameter of the dnSmpl actor is constrained to take on a value less than the factor parameter, and this is reflected in the actor domain, $domain(\text{dnSmpl})$ which is given by

$$\{\{(\text{factor}, x), (\text{phase}, y)\} | (x \in domain(\text{factor}), y \in domain(\text{phase}), y < x)\}. \quad (11)$$

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

The *port consumption function* associated with A, denoted

$$\kappa_A : (in(A) \times \overline{domain(A)}) \rightarrow Z^+, \quad (12)$$

gives the number of tokens consumed from a specified input port on each invocation of actor A. For any complete configuration C and any input port $\theta \in in(A)$, $\kappa_A(\theta, C)$ gives the number of tokens consumed from port θ if for each $p \in params(A)$, parameter p of A is assigned the value $C(p)$.

The *port production function*

$$\varphi_A : (out(A) \times \overline{domain(A)}) \rightarrow Z^+ \quad (13)$$

associated with A is defined in a similar fashion. If the actor A is understood from context, we may simply write $\kappa(\varphi)$ in place of $\kappa_A(\varphi_A)$.

In general, a software subroutine is provided that takes as input a complete configuration C, outputs a boolean value indicating whether or not $C \in domain(A)$, and also outputs the values $\{(\rho, \kappa_A(\rho, C)) \big| (\rho \in in(A))\}$ and

$\{(\rho, \varphi_A(\rho, C)) \mid (\rho \in out(A))\}$ [1]. We refer to this subroutine as the *parameter interpretation function f_A of A*.

In order to facilitate practical implementations of applications specified in PSDF, a *max token transfer function* is associated with A, denoted as

$$\tau_A : (in(A) \cup out(A)) \rightarrow Z^+, \tag{14}$$

which specifies an upper bound on the maximum number of tokens transferred (produced or consumed) at a port of actor A. This maximum value is necessary to ensure bounded memory executions of consistent PSDF specifications. The concept of bounding the maximum token transfer at an actor port appears similar to BDDF [28], but our use is very different, as we will discuss in Section 3.8.2.

From the above definitions, a complete configuration $C \in \overline{domain(A)}$ for a PSDF actor A yields a (pure) SDF actor, which we denote by $instance_A(C)$. On each invocation of $instance_A(C)$, the constant number $\kappa_A(\theta, C)$ of tokens is consumed from each input port θ, and the constant number $\varphi_A(\theta, C)$ of tokens is produced onto each output port θ. This SDF actor $instance_A(C)$ is called an *SDF instance* of the PSDF actor A. Thus a PSDF actor can be viewed as a mapping from $\overline{domain(A)}$ into the set of SDF actors.

In summary, a PSDF actor A has nine attributes — the parameter set $params(A)$; the parameter domain $domain(A)$; the configuration $config_A$, which

1. In some cases, this method for processing parameters can be streamlined to reduce the overall run-time overhead of parameter interpretation. We discuss this further in Section 3.7. However, this streamlining does not conflict with our techniques for consistency analysis and scheduling. Thus, in our analysis, we adopt this simpler formalization for clarity.

may be complete or incomplete, but must be valid; the set of input ports $in(A)$; the set of output ports $out(A)$; the port consumption and production functions κ_A and φ_A, which specify the dataflow properties of the actor; the parameter interpretation function f_A, which provides implementations of the functions κ_A and φ_A; and the max token transfer function τ_A, which specifies an upper bound on the tokens transferred at an actor port. Other important properties, such as the actor execution time and the internal memory requirement, may also be represented and used in a parameterized fashion (e.g., for optimization purposes). In this report, we focus only on properties that are essential to the operational semantics of PSDF.

From the above definitions, a PSDF actor that has a nonempty parameter set, but has no valid, complete configurations is pathological. Such an actor is not usable in any practical sense. Thus, a *valid* PSDF actor is one that satisfies $params(A) = \varnothing$, or $|\overline{domain(A)}| \geq 1$. Furthermore, we say that a PSDF actor A is a *(pure) SDF* actor if $params(A) = \varnothing$, or $|domain(A)| = 1$. We also refer to such actors as *type 1 PSDF actors*.

Now, consider those PSDF actors that satisfy $params(A) \neq \varnothing$, and $|\overline{domain(A)}| \geq 1$, but whose port consumption and production quantities ($\kappa_A(\theta, *)$ and $\varphi_A(\theta, *)$) are all known to be constant over their respective parameter domains. We refer to these as *type 2 PSDF actors*. At first, this may also seem to be another pathological class of actors, but in reality, such actors are very common.

Example 6: An IIR filter actor in Ptolemy [12] implements an infinite impulse response filter with the transfer function

$$H(z) = (g \times N(1/z))/(D(1/z)) \quad (15)$$

The actor has three parameters

$$params(\text{IIR}) = \{\text{gain, numerator, denominator}\}, \quad (16)$$

where *gain* specifies g, and the floating point arrays *numerator* and *denominator* specify $N(\ )$ and $D(\ )$ respectively. The actor has one input port *in*, and one output port *out*, and it always consumes one token from its input port, and produces one token on its output port. In other words, $\kappa_A(\text{in}, *) = 1$, and $\varphi_A(\text{out}, *) = 1$ for any valid configuration of $params(\text{IIR})$.

Type 2 actors differ from pure SDF actors in that their functionality is parameterized even though their dataflow behavior is not. Since dataflow behavior is not affected, the distinction between type 1 and type 2 actors has largely been ignored in past work on analysis, scheduling, and optimized implementation of SDF graphs.

An exception is the *code sharing optimization* developed by Sung, Kim, and Ha [32], which attempts to implement multiple parameterizations of the same actor definition (or multiple copies of the same type 1 actor definition) with the same sequence of program memory instructions. The objective of this technique is to reduce memory requirements of type 1 and type 2 actors whose core functionality is replicated within a specification.

The distinction between type 1 and type 2 actors primarily arises in issues of syntax. Since type 2 actors provide great convenience without complicating the

compilation process, DSP programming environments that employ SDF typically incorporate such actors. Previous analysis and optimization techniques for SDF graphs, such as those developed in [1, 4, 33, 36], apply to specifications that contain arbitrary combinations of type 1 and type 2 PSDF actors.

A *type 3 PSDF actor* is one that has a nonempty parameter set, and whose port production or consumption quantities may vary depending on the parameter setting. This possibility significantly complicates the problem of compiling PSDF specifications. Indeed, the design of our PSDF representation paradigm is centered around addressing this complication in an efficient manner. In exchange for accommodating this complication, we obtain a model that has much higher expressive power than what is offered by type 1 and type 2 PSDF actors alone.

3.3. PSDF Edges

Like a PSDF actor, a *PSDF edge* e also has an associated parameter set, parameter domain, and configuration. These are denoted by $params(e)$, $domain(e)$, and $config_e$, respectively. The set $\overline{domain(e)}$ is defined in a manner analogous to $\overline{domain(A)}$.

Connectivity information of a PSDF edge e is specified through two attributes $src(e)$ and $snk(e)$. The value of $src(e)$ must be an output port of a some PSDF actor, and $snk(e)$ must be an input port of some PSDF actor. Note that in this case we are overloading the directed multigraph definition of an edge e (Section 1.1) that is used in SDF, where $src(e)$ and $snk(e)$ refer to the source actor and sink actor, respectively, of e. In case of PSDF also, we will sometimes use the latter def-

inition of $src(e)$ and $snk(e)$, and which definition we actually mean will become clear from context.

Apart from connectivity, a PSDF edge can have three other characteristics — the amount of delay on the edge (the number of initial tokens, that are referred to as *delay tokens*), the initial value of each delay token, and the number of invocations (*re-initialization period*) of the sink actor (i.e. the actor to which the port $snk(e)$ belongs) after which each delay token has to be re-initialized with its initial value. Different configurations of the parameter set of a PSDF edge may allow for a range of different values for each of these three quantities.

The *delay function* $\delta_e : \overline{domain(e)} \rightarrow \aleph$ associated with e gives the delay on that edge that results from any valid parameter setting. For every delay token α on edge e, the *delay initial value function* $\upsilon_e : (\overline{domain(e)} \times \alpha \rightarrow \Re)$ gives the initial value of a specified delay token for a valid parameter configuration of e. The *delay re-initialization period function* $\gamma_e : (\overline{domain(e)} \times \alpha \rightarrow \aleph)$ specifies the re-initialization period of a delay token for any valid configuration of the parameter set of e. The *parameter interpretation function* f_e of e provides implementations of δ_e, υ_e, and γ_e in a manner analogous to the parameter interpretation function of a PSDF actor.

When the notation is not ambiguous, we may suppress the attributes $params(e)$, δ_e, υ_e, ρ_e, $domain(e)$, f_e and simply denote e by the ordered pair $(src(e), snk(e))$, specifying its connectivity information. If $params(e) \neq \varnothing$, and δ_e is not constant over $\overline{domain(e)}$, then a *max delay value*, denoted $\mu_e \in \aleph$ must

be specified for e, which provides an upper bound on the maximum number of delay tokens that can reside at any time on e. This bound is necessary to ensure that there are no unbounded token accumulation on the PSDF edge e across invocations of the PSDF graph to which e belongs, leading to bounded memory executions of consistent PSDF specifications. These issues are discussed in more detail in Sections 3.6 and 3.8.

3.4. PSDF Graphs

A *PSDF graph* G is an ordered pair (V_G, E_G), where V_G is a set of PSDF actors, and E_G is a set of PSDF edges that connect a subset of the actor output ports to a subset of the set of input ports. More precisely,

$$\text{for each } e \in E_G,\ src(e) \in IN(V_G), \text{ and } snk(e) \in OUT(V_G), \tag{17}$$

where

$$IN(V_G) \equiv \{in(v) | (v \in V_G)\}, \text{ and } OUT(V_G) \equiv \{out(v) | (v \in V_G)\}. \tag{18}$$

Furthermore, no two edges share a common originating or terminating port:

$$\text{if } e_1 \text{ and } e_2 \text{ are distinct members of } E_G, \text{ then } src(e_1) \neq src(e_2), \text{ and}$$

$$snk(e_1) \neq snk(e_2). \tag{19}$$

If (17-19) do not all hold, then the ordered pair (V_G, E_G) is not a PSDF graph.

Given a PSDF graph (V_G, E_G), we denote the set of *internally-connected*

input ports $\{snk(e)|(e \in E_G)\}$ and *internally-connected output ports* $\{src(e)|(e \in E_G)\}$ by I_G and O_G, respectively. The sets I_G and O_G may be proper subsets of $IN(V_G)$ and $OUT(V_G)$. Each member of $(IN(V_G) - I_G)$ is called an *interface input* of G, and similarly, each member of $(OUT(V_G) - O_G)$ is called an *interface output* of G. The sets of interface inputs and outputs of G are denoted $inputs(G)$ and $outputs(G)$, respectively.

The *parameter set* of G is defined by

$$params(G) = Aparams(G) \cup Eparams(G), \tag{20}$$

where

$$Aparams(G) = \{(A, p)|(A \in V_G, params(A) \neq \varnothing, config_A(p) = \perp)\}, \tag{21}$$

and

$$Eparams(G) = \{(e, p)|(e \in E_G, params(e) \neq \varnothing, config_e(p) = \perp)\}. \tag{22}$$

Each $(A, p) \in Aparams(G)$, called an *actor parameter* of G, is a parameter of G with $domain((A, p)) = domain(p)$. Similarly, each $(e, p) \in Eparams(G)$ is called an *edge parameter of* G, and has $domain((e, p)) = domain(p)$. The *set of unspecified parameters of actor* A *in* G is defined as

$params_A(G) = \{(A, p)|(params(A) \neq \varnothing)$ **and** $(config_A(p) = \perp)\}$, and is empty if

$$params(A) = \varnothing. \tag{23}$$

The *set of unspecified parameters of edge e in G* is similarly defined as $params_e(G)$,

$params_e(G) = \{(e, p) | (params(e) \neq \varnothing) \textbf{and} (config_e(p) = \perp)\}$, and is empty if

$$params(e) = \varnothing. \tag{24}$$

For a nonempty parameter set, each member of $params(G)$ is a *parameter* of G. A *configuration of G* is a configuration of $params(G)$. The domain of $params(G)$, denoted $domain(G)$, is the joint domain $Jdomain(params(G))$ (see (7)), and $\overline{domain(G)}$ denotes the set of complete configurations in $domain(G)$. Intuitively, $params(G)$ is the set of PSDF actor and edge parameters in G whose values are left unspecified by the application programmer.

The *simplified PSDF graph* associated with (V_G, E_G) is the directed multi-graph (V_G, E'), where

$$E' = \{(actor(src(e)), actor(snk(e))) | (e \in E_G)\}. \tag{25}$$

The simplified graph is the most commonly used representation for analyzing pure SDF graphs.

3.4.1 The Parameterized Repetitions Vector

Suppose that $G = (V_G, E_G)$ is a PSDF graph, and $\bar{p} \in \overline{domain(G)}$ is a complete configuration of G. Then, clearly, a pure SDF graph emerges by "applying" the configuration $\bar{p}$ to unspecified actor and edge parameters in G, producing complete configurations for each actor and edge parameter in G. For a PSDF actor

A in G, we define the *instantiated configuration of A in G associated with the complete configuration $\bar{p}$* as

$$config_{A,\bar{p}} \equiv rfmt(config_A, \bar{p} | params_A(G)) \tag{26}$$

Similarly, for a PSDF edge e in G, the *instantiated configuration of e in G associated with the complete configuration $\bar{p}$* is defined as

$$config_{e,\bar{p}} \equiv rfmt(config_e, \bar{p} | params_e(G)) \tag{27}$$

The SDF graph that results from the complete configuration $\bar{p}$, is called the *instance of G associated with the complete configuration $\bar{p}$,* and it is denoted by $instance_G(\bar{p})$. In precise terms, we have

$$instance_G(\bar{p}) = (V', E'), \tag{28}$$

where

$$V' = \{instance_A(config_{A,\bar{p}}) \,|\, (A \in V_G)\}, \tag{29}$$

and

$$E' = \{instance_e(config_{e,\bar{p}}) \,|\, (e \in E_G)\}, \tag{30}$$

If the instantiated SDF graph $instance_G(\bar{p})$ is sample rate consistent, then it is possible to compute the *parameterized repetitions vector* $\mathbf{q}_{G,\bar{p}}$, indexed by the actors in G, with respect to the complete configuration $\bar{p}$, such that $\mathbf{q}_{G,\bar{p}}$ satisfies

the *configured balance equations* for every edge e in $instance_G(\bar{p})$:

$$\mathbf{q}_{G,\bar{p}}(\mathrm{SC}) \times \varphi_{\mathrm{SC}}(src(e), config_{\mathrm{SC},\bar{p}}) = \mathbf{q}_{G,\bar{p}}(\mathrm{SN}) \times \kappa_{\mathrm{SN}}(snk(e), config_{\mathrm{SN},\bar{p}}) \quad (31)$$

where $\mathrm{SC} = actor(src(e))$ is the actor at the source of e, and

$\mathrm{SN} = actor(snk(e))$ is the actor at the sink of e.

More precisely, if the configured balance equations for $instance_G(\bar{p})$ have a positive integer solution, then there exists a unique, minimal positive integer solution which is given by $\mathbf{q}_{G,\bar{p}}$. If a positive, integer solution does not exist, then $\mathbf{q}_{G,\bar{p}}$ does not exist.

In the context of the simplified PSDF graph (Section 3.4), we will use notation in accordance with SDF graphs, as described in Section 1.2. In particular, given a configuration $\bar{p} \in \overline{domain(G)}$ of the PSDF graph G, in the simplified PSDF graph, we will use $p(e)$ to refer to the number of tokens produced onto PSDF edge e, which is the same as $\varphi_{actor(src(e))}(src((e), config_{actor(src(e)),\bar{p}}))$; $c(e)$ to refer to the number of tokens consumed from PSDF edge e, which is the same as $\kappa_{actor(snk(e))}(snk((e), config_{actor(snk(e)),\bar{p}}))$; and $d(e)$ to refer to the number of delay tokens on PSDF edge e, which is the same as $\delta_e(config_{e,\bar{p}})$. Suppressing the argument $\bar{p}$ does not cause ambiguity in the context in which we use $p(e)$, $c(e)$, and $d(e)$. Finally, we will use $src(e)$ and $snk(e)$ to refer to the source and sink actors of PSDF edge e, instead of source and sink actor ports of e. In this context, the max token transfer bound at an actor port can be interpreted as a bound on the number of tokens produced ($p(e)$) onto or consumed ($c(e)$) from the PSDF edge e

incident at that actor port.

3.5. PSDF Specifications

A PSDF specification Φ contains three PSDF graphs — the *init graph* Φ_i, the *subinit graph* Φ_s, and the *body graph* Φ_b. Intuitively, Φ_i is invoked once at the beginning of an invocation (a minimal periodic invocation) of the hierarchical "parent" graph of Φ 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. Parameter values of Φ_b and Φ_s that remain constant throughout an execution of the parent subsystem are computed by Φ_i or "passed on" by Φ_i from subsystem parameter settings. Parameter values of Φ_b that remain constant throughout each invocation of Φ, but can change across invocations are set up by Φ_s.

The set of interface outputs $outputs(\Phi_i)$ of the init graph can be partitioned into two disjoint subsets:

$$outputs(\Phi_i) = ToBody(\Phi_i) \cup ToSubinit(\Phi_i), \tag{32}$$

where $ToBody(\Phi_i)$ is the set of output ports used to set parameter values in the body graph, and $ToSubinit(\Phi_i)$ is the set of output ports used to set parameter values in the subinit graph. The function

$$\Phi_{tb} : ToBody(\Phi_i) \rightarrow params(\Phi_b) \tag{33}$$

associates each interface output in $ToBody(\Phi_i)$ with the body graph parameter that is controlled by it. Similarly, the function

$$\Phi_{\text{ts}} : ToSubinit(\Phi_i) \rightarrow params(\Phi_s) \tag{34}$$

specifies which parameter of Φ_s is set by each member of $ToSubinit(\Phi_i)$. Each invocation of Φ_i must produce *exactly* one token on each member of $outputs(\Phi_i)$. Such PSDF-specific issues of sample-rate consistency and local synchrony are examined further in Section 3.6.

Interface outputs of the subinit graph are used exclusively to set parameters in the body graph. Thus, we have a mapping

$$\Phi_{\text{fs}} : outputs(\Phi_s) \rightarrow params(\Phi_b) \tag{35}$$

that associates each interface output of Φ_s with the body graph parameter that is configured by it. The "fs" subscript in (35) stands for "from subinit." Each invocation of Φ_s is also constrained to produce exactly one token on each member of $outputs(\Phi_s)$.

The mappings Φ_{tb} and Φ_{fs} must satisfy

$$(\Phi_{\text{tb}}(ToBody(\Phi_i)) \cap \Phi_{\text{fs}}(outputs(\Phi_s))) = \varnothing, \text{ and}$$

$$\Phi_{\text{tb}}(ToBody(\Phi_i)) \cup \Phi_{\text{fs}}(outputs(\Phi_s)) = params(\Phi_b). \tag{36}$$

That is, the images of Φ_{tb} and Φ_{fs} must partition the parameter set of the body graph.

A specification Φ also has a (possibly empty) set of inputs, denoted $inputs(\Phi)$, and a (also possibly empty) set of outputs, denoted $outputs(\Phi)$. These inputs and outputs correspond to endpoints of dataflow connections (edges) when Φ is embedded within a larger subsystem. (Unspecified) parameter values of the subinit graph may be bound to dataflow inputs of Φ, and the subinit graph may also receive dataflow inputs from Φ. In the former case, the values of input tokens are set as the values of designated graph parameters, and in the latter case, tokens are directed to the inputs of computational actors in Φ_s. Thus, there are three types of inputs to Φ — inputs that are bound to parameters of Φ_s (*parameter inputs*), inputs that are dataflow inputs to Φ_s (*subinit inputs*), and inputs that are dataflow inputs to Φ_b (*body inputs*). These three subsets of $inputs(\Phi)$ are denoted $inputs_p(\Phi)$, $inputs_s(\Phi)$, and $inputs_b(\Phi)$, respectively. The mapping

$$\Phi_{\text{ps}} : inputs_p(\Phi) \rightarrow params(\Phi_s) \tag{37}$$

specifies which parameter is set by each parameter input. Thus,

$$\Phi_{\text{ps}}(inputs_p(\Phi)) \cap \Phi_{\text{ts}}(ToSubinit(\Phi_i)) = \varnothing; \tag{38}$$

$$inputs_s(\Phi) = inputs(\Phi_s);\ inputs_b(\Phi) = inputs(\Phi_b); \tag{39}$$

$$inputs(\Phi) = inputs_p(\Phi) \cup inputs_s(\Phi) \cup inputs_b(\Phi). \tag{40}$$

The outputs of Φ are simply the outputs of the associated body graph.

$$outputs(\Phi) = outputs(\Phi_b). \quad (41)$$

All parameters of Φ_i, and those parameters of Φ_s contained in the set

$$params_{\Phi_S}(\Phi) = params(\Phi_s) - (\Phi_{ps}(inputs_p(\Phi)) \cup \Phi_{ts}(ToSubinit(\Phi_i))) \quad (42)$$

are parameters of Φ. Thus, the *parameter set* of Φ, denoted $params(\Phi)$, is defined by

$$params(\Phi) \equiv params(\Phi_i) \cup params_{\Phi_S}(\Phi). \quad (43)$$

The parameters of Φ are configured (assigned values) in init and subinit graphs of hierarchically higher-level subsystems, and we refer to this mechanism of parameter value passing as *initflow* to distinguish it from dataflow. Thus, the parameters of Φ_s ($params_{\Phi_S}(\Phi)$) that appear as a part of the parameters of Φ are termed the *initflow parameters* of Φ_s. $Aparams(\Phi)$ and $Eparams(\Phi)$ respectively denote the subsets of actor and edge parameters in $params(\Phi)$. A *configuration of* Φ is a configuration of $params(\Phi)$. The domain of $params(\Phi)$, denoted $domain(\Phi)$, is the joint domain $Jdomain(params(\Phi))$ (see (7)), and $\overline{domain(\Phi)}$ denotes the set of complete configurations in $domain(\Phi)$.

In other words, a parameter of Φ is a parameter of the associated init graph, or a parameter of the associated subinit graph that is not bound to an input of Φ, nor to an output of the init graph.

3.5.1 Hierarchical Actors

In designing an application of any reasonable complexity, abstraction of portions of the design into a single elementary block (actor) is necessary for scalability and modularity. Thus, all block diagram DSP programming environments provide the concept of a hierarchical block in some form, where the internals of the block is described by another dataflow graph. In our context, we refer to this kind of hierarchy as *syntactic hierarchy*. As mentioned in Section 2.1, syntactic hierarchy is allowed in PSDF through *subblocks*, and dealt with in the usual way by flattening the system and replacing the subblock with the dataflow graph that it represents. Thus, syntactic hierarchy has no connotations on the semantics of the dataflow model like scheduling decisions, and simply serves as a convenient tool for the application programmer.

As presented in Sections 3.1 to 3.5, our parameterized model delivers additional expressive power to the application programmer by providing PSDF subsystems (distinct from subblocks) that can be used to represent logical functional units capable of configuring their internal functional and dataflow behavior. External handles are available as subsystem parameters that can be used to control subsystem behavior from outside the subsystem. This naturally leads to the concept of a hierarchical representation, where subsystem parameters are configured by its ancestor subsystems. In developing support for such a hierarchy, it is particularly desirable that from above, the "child" subsystem appears just as a PSDF actor. Thus, the PSDF model incorporates the concept of a *hierarchical PSDF actor* that repre-

sents a child PSDF subsystem (specification). This hierarchy is very different from syntactic hierarchy in the sense that this arises as a direct consequence of the PSDF semantics — indeed, it is one of the key ingredients in the PSDF semantic model. We refer to this hierarchy as *semantic hierarchy* or *control hierarchy*, and henceforth whenever we refer to hierarchical actors, we do so in the context of semantic hierarchy. Semantic hierarchy plays an important role in the PSDF operational semantics, as we will discuss in Section 3.7. In particular, unlike syntactic hierarchy, semantic hierarchy is never flattened, instead, its presence is factored into the scheduling and consistency rules.

Formally, a *hierarchical PSDF actor*, or simply a *hierarchical actor*, H is a PSDF actor that has an *associated subsystem,* $subsystem(H)$, which is a PSDF specification. The input and output ports of H are in one-to-one correspondence with the inputs and outputs of $subsystem(H)$. The bijective mapping

$$g_H : (in(H) \cup out(H)) \rightarrow (inputs(subsystem(H)) \cup outputs(subsystem(H))), \quad (44)$$

defines this correspondence.

An illustration of the coordination mechanism between the init graph, subinit graph and body graph in a PSDF specification Φ represented by the hierarchical actor H is given in Fig. 14. The figure shows a hierarchical PSDF actor H as it appears externally, and the internals of the specification Φ represented by H. A bold 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

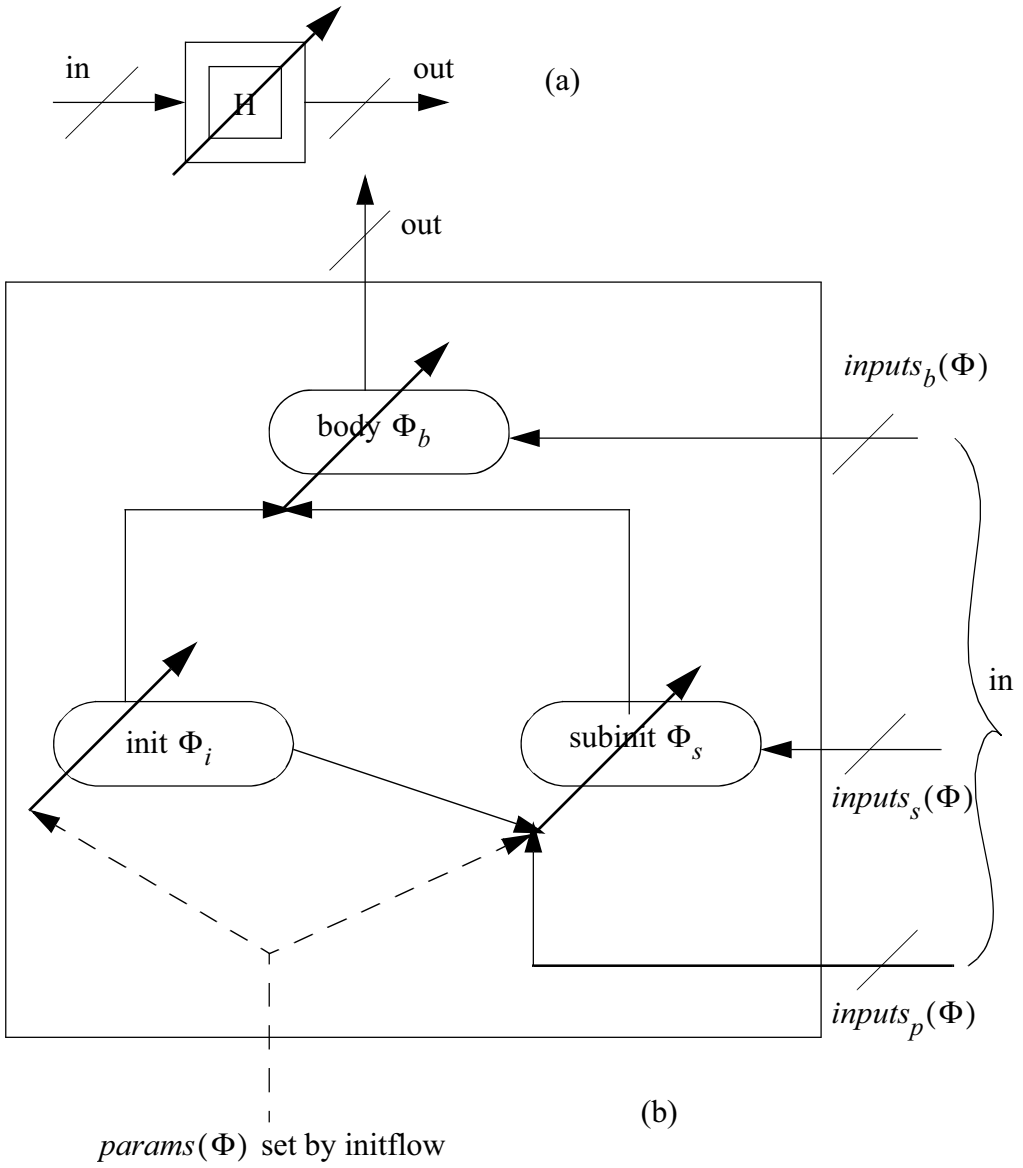


Figure 14. The operational structure of a PSDF specification.

edges. Dataflow is denoted by bold lines, while initflow is denoted by dashed lines.

The parameter set of H, the domain of H, and the set of complete configurations of H is given by

$$params(H) = params(\Phi);\ domain(H) = domain(\Phi);\ \text{and}$$

$$\overline{domain(H)} = \overline{domain(\Phi)}; \tag{45}$$

where $\Phi = subsystem(H)$. In Sections 3.7, and 4.3, we will discuss techniques to determine the production and consumption functions κ_H and φ_H, and to derive the parameter interpretation function f_H. In general, the max token transfer function τ_H should also be derived from the memory requirements of the subinit and body graphs of the subsystem that H represents. Deriving such tight memory bounds is a complex issue, and warrants further investigation. At present, we require that the application programmer explicitly specify a max token transfer function τ_H for each hierarchical actor H.

An actor that is not a hierarchical actor is called a *leaf* actor. In general, PSDF graphs may contain both hierarchical and leaf actors. If A is an actor in the PSDF graph G, we say that A is *immediately nested* in G. If A is hierarchical, we also say that $subsystem(A)$ is immediately nested in G. If $\Phi = subsystem(A)$, we say that Φ is a *child* subsystem of G, and G is called the *parent graph* of Φ. The enclosing subsystem of G with which G is associated as an init, subinit or body graph is called the *parent subsystem* (or *parent specification*) of G. If G does not contain any hierarchical actors, it is referred to as a *leaf* graph. In a *leaf* subsystem

Φ, each of Φ_i, Φ_s, and Φ_b is a leaf graph. Actor A is *nested in* G *at nesting depth* d if there is a sequence of PSDF specifications G_1, G_2, …, G_d such that A is immediately nested in G_1, and each G_i is immediately nested in G_{i+1}. Similarly, $subsystem(A)$ is nested in G at nesting depth d if A is nested in G at nesting depth d.

The topmost PSDF specification Φ is implicitly assumed to be embedded in a PSDF graph with a single hierarchical actor that represents Φ.

3.6. Local Synchrony in PSDF

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

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

A PSDF graph G is locally synchronous, if for every $\bar{p} \in \overline{domain(G)}$, the instantiated SDF graph $instance_G(\bar{p})$ has the following properties: it is sample rate consistent (i.e. $\mathbf{q}_{G,\bar{p}}$ exists); it is deadlock free; the max token transfer bound is satisfied for every port of every actor; the max delay value bound is satisfied for every edge; and every child subsystem is locally synchronous. Formally, the *local synchrony condition* for a PSDF graph $G = (V, E)$ is

for each $\bar{p} \in \overline{domain(G)}$, $instance_G(\bar{p})$ has a valid schedule; for each actor $v \in V$, for each input port $\phi \in in(v)$, $\kappa_v(\phi, config_{v,\bar{p}}) \le \tau_v(\phi)$, and for each output port $\phi \in out(v)$, $\varphi_v(\phi, config_{v,\bar{p}}) \le \tau_v(\phi)$; for each edge $e \in E$, if μ_e exists, $\delta_e(config_{e,\bar{p}}) \le \mu_e$; for each hierarchical actor H in G, $subsystem(H)$ is locally synchronous. (46)

If (46) is satisfied for every $\bar{p} \in \overline{domain(G)}$, then G is *inherently locally synchronous* (or simply *locally synchronous*). If no $\bar{p} \in \overline{domain(G)}$ satisfies (46), then G is *inherently locally synchronous* (or simply *locally non-synchronous*). If G

is neither inherently locally synchronous, nor inherently locally non-synchronous, then G is *partially locally synchronous* (i.e. there exists at least one $\bar{p} \in \overline{domain(G)}$ for which (46) is satisfied, and there exists at least one $\bar{p} \in \overline{domain(G)}$ for which (46) is not satisfied). We sometimes separately refer to the different components of (46) as *dataflow consistency* (existence of a valid schedule), *bounded memory consistency* (upper bounds are satisfied for each actor port and each edge), and *subsystem consistency* (each subsystem is locally synchronous) of the PSDF graph G.

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

First, the init graph must produce exactly one token on each output port on each invocation. This is because each output port is bound to a parameter setting (of the body graph or subinit graph). An alternative is to allow multiple tokens to be produced on an init output port, and assign those values one by one to the dependent parameter on successive invocations of Φ. But this leads to two problems. First, we would have to line up the number of tokens produced with the number of invocations of Φ, thus giving rise to sample rate consistency issues across graph bound-

aries, which needlessly complicates the semantics. Second, it violates the principle that parameters set in init maintain constant values throughout one invocation of the parent graph of Φ, which in turn violates the requirements for local SDF scheduling. The interface dataflow of the hierarchical actor representing Φ is allowed to depend on parameters set in the init graph. For the parent graph of Φ to be configured as an SDF graph on every invocation, each such embedded hierarchical actor must behave as an SDF actor, for which the parameters set in init must remain constant throughout an invocation of the parent graph.

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

Third, the number of tokens consumed by the subinit graph from each input port in $inputs_s(\Phi)$ must be not be a function of the subinit graph parameters that are bound to dataflow inputs of Φ. Finally, the number of tokens produced or consumed at each specification interface port of the body graph (each member of $(inputs_b(\Phi) \cup outputs(\Phi))$) must be a function of the body graph parameters that are controlled by the init graph. The third and fourth conditions ensure that a hierarchically nested PSDF specification behaves like an SDF actor throughout any single

invocation of the parent graph in which it is embedded, which is necessary for local SDF scheduling.

In mathematical terms, the first condition is the requirement that

$$\text{the init graph } \Phi_i \text{ is locally synchronous;} \tag{47}$$

$$\text{for each } \bar{p} \in \overline{domain(\Phi_i)}\text{, and each interface output port } \phi \text{ of } \Phi_i\text{,}$$

$$\mathbf{q}_{\Phi_i, \bar{p}}(actor(\phi)) = \varphi_{actor(\phi)}(\phi, config_{actor(\phi), \bar{p}}) = 1\,. \tag{48}$$

(47) and (48) comprise the *init condition* for local synchrony of Φ, which says that the init graph must be (inherently) locally synchronous and must produce exactly one token at each interface output port on each invocation. Similarly, the second and the third conditions are the requirement that

$$\text{the subinit graph } \Phi_s \text{ is locally synchronous;} \tag{49}$$

$$\text{for each } \bar{p} \in \overline{domain(\Phi_s)}\text{, and each interface output port } \phi \text{ of } \Phi_s\text{,}$$

$$\mathbf{q}_{\Phi_s, \bar{p}}(actor(\phi)) = \varphi_{actor(\phi)}(\phi, config_{actor(\phi), \bar{p}}) = 1\,; \tag{50}$$

$$\text{for each interface input port } \phi \text{ of } \Phi_s\text{, the product}$$

$$\mathbf{q}_{\Phi_s, \bar{p}}(actor(\phi))\kappa_{actor(\phi)}(\phi, config_{actor(\phi), \bar{p}}) \text{ is invariant over } \Phi_{\text{ps}}(inputs_p(\Phi))\text{,}$$

$$\text{where } \bar{p} \in \overline{domain(\Phi_s)}\,. \tag{51}$$

We refer to (49) and (50) as the *subinit output condition*, and to (49) and (51) as the *subinit input condition* for local synchrony of Φ. Thus, the subinit graph must be (inherently) locally synchronous, Φ_s must produce only one token at each of its interface output ports on each invocation, and the number of tokens consumed from an input port of Φ_s (during an invocation of Φ) must be a function only of the parameters that are controlled by the init graph, and the initflow parameters of Φ_s (see (42)).

Finally, the fourth condition requires that

the body graph Φ_b is locally synchronous; (52)

for each interface input port ϕ of Φ_b, the product

$\mathbf{q}_{\Phi_b, \bar{p}}(actor(\phi))\kappa_{actor(\phi)}(\phi, config_{actor(\phi), \bar{p}})$ is invariant over $\Phi_{\text{fs}}(outputs(\Phi_s))$,

where $\bar{p} \in \overline{domain(\Phi_b)}$; (53)

for each interface output port ϕ of Φ_b, the product

$\mathbf{q}_{\Phi_b, \bar{p}}(actor(\phi))\varphi_{actor(\phi)}(\phi, actor(\phi))$ is invariant over $\Phi_{\text{fs}}(outputs(\Phi_s))$,

where $\bar{p} \in \overline{domain(\Phi_b)}$. (54)

(52), (53) and (54) are termed the *body condition* for local synchrony of Φ. In other words, the body graph must be (inherently) locally synchronous, and the total num-

ber of tokens transferred at any port of Φ_b throughout a given invocation of Φ must depend only on those parameters of Φ_b that are controlled by Φ_i.

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

If (47), (49), (52) hold (each of the graphs Φ_i, Φ_s, and Φ_b is inherently locally synchronous), and (48) holds for each $\bar{p} \in \overline{domain(\Phi_i)}$ (the number of tokens transferred at each interface output of init is one), and (50) holds for each $\bar{p} \in \overline{domain(\Phi_s)}$ (the number of tokens transferred at each interface output of subinit is one), and (51), (53), and (54) all hold (tokens transferred at the interface inputs of subinit, and at the interface ports of body are invariant over the subinit and body graph parameters that are not set in init), then Φ is *inherently locally synchronous* (or simply *locally synchronous*). If either of the graphs Φ_i, Φ_s, and Φ_b is locally non-synchronous, or no $\bar{p} \in \overline{domain(\Phi_i)}$ satisfies (48), or no $\bar{p} \in \overline{domain(\Phi_s)}$ satisfies (50), then Φ is *inherently locally non-synchronous* (or simply *locally non-synchronous*). If Φ is neither inherently locally synchronous, nor inherently locally non-synchronous, then Φ is *partially locally-synchronous*. According to the definition of invariancy (see (6)), (51) can be verified by computing the tokens consumed at each interface input port of Φ_s for each $\bar{p} \in \overline{domain(\Phi_s)}$, and checking if the tokens consumed are identical for each pair $\overline{p_1}, \overline{p_2} \in \overline{domain(\Phi_s)}$, for which $\overline{p_1}\Big|(params(\Phi_s) - \Phi_{ps}(inputs_p(\Phi)))$ is equal to

$\overline{p_2} \mid (params(\Phi_s) - \Phi_{ps}(inputs_p(\Phi)))$. Similar verification can be applied to (53), and (54). Note that if any of the invariancy conditions (51, 53, 54) do not hold, then that does not lead to local non-synchrony of Φ, but only to partial local synchrony of Φ.

3.7. Operational Semantics

Based on the formalization introduced in Sections 3.1 to 3.6, we develop in this section a precise operational semantics for PSDF. Fig. 15 gives a pseudocode description. Executing a PSDF specification is equivalent to executing the graph in which the topmost specification is embedded according to the rules specified in the routine *execute*. Given a graph G and a complete configuration C for the parameters of the graph, this routine verifies local synchrony of G, computes a schedule for G and executes that schedule. In case of a local synchrony violation, execution is terminated. The routine returns a configuration C_{out} of the parameters that are configured at the interface output ports of G, as a result of executing G. For example, if G is an init graph of a specification, then in general it will set some parameters of the body graph and the subinit graph at its interface output ports. The configuration of these parameters will be returned after executing G.

To compute a schedule for G, we need to know the interface token flow of every hierarchical actor H, which might in general depend on the internals of the subsystem represented by H. In the operational semantics shown in Fig. 15, for every hierarchical actor H present in G, first a configuration is determined for the init graph parameters of the subsystem Φ represented by H. Any parameters of the

```
function execute(graph G, configuration C)
    foreach hierarchical actor H in G
        Φ = subsystem(H)
        C_Φi = configure_graph(Φ_i, C)
        C_Φinit-set = execute(Φ_i, C_Φi)
        C_Φs = configure_graph(Φ_s, C ∪ C_Φinit-set)
        C_Φb = configure_graph(Φ_b, C_Φinit-set)
        precompute_interface_token_flow(Φ_s, C_Φs)
        precompute_interface_token_flow(Φ_b, C_Φb)
    end for
    configure_as_SDF(G, C)
    compute_repetitions_vector(G); S = compute_schedule(G)
    configuration C_internal = ∅; configuration C_out = ∅
    while ((L = get_next_firing(S)) ≠ NULL)
        if (L is a hierarchical actor)
            Φ = subsystem(L)
            C_Φs = configure_graph(Φ_s, C ∪ C_Φinit-set ∪ C_internal)
            C_Φsubnit-set = execute(Φ_s, C_Φs)
            C_Φb = configure_graph(Φ_b, C_Φinit-set ∪ C_Φsubnit-set)
            execute(Φ_b, C_Φb)
            verify_interface_token_flow(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)
                    C_out = C_out ∪ {p, v}
                else
                    if ({p, *} ∈ C_internal) C_internal = C_internal − {p, *}
                    C_internal = C_internal ∪ {p, v}
                end if
            end if
        end if
    end while
    return C_out
end function
```

Figure 15. The operational semantics of a PSDF specification.

init graph will also occur in the parameter set of G, hence a complete configuration for the init graph can be determined from the complete configuration C of G. The init graph is then executed with this complete configuration by recursively calling the routine *execute*. After execution of the init graph, a configuration of the subinit and body graph parameters of Φ that are configured in the init graph, is returned in $C_{\Phi_{\text{init-set}}}$. Based on the configuration of G, and the configuration determined by the init graph of Φ, a configuration is computed for the subinit and body graphs of Φ. Those parameters of the subinit graph that are bound to dataflow inputs of Φ, and those parameters of the body graph that are set in the subinit graph, have unknown values, and are not present in the known configurations determined so far. These parameters are assigned their default values, and an unspecified default value (for any of the unknown parameters) is detected as invalid. The routine for computing a complete configuration for the parameters of a graph G, given a complete configuration of a set of parameters that may include some of the parameters of G, is shown separately in Fig. 16. The configurations of the subinit and body graph of Φ, computed as above, are then used to pre-compute the token flow at the interface ports of the subinit and body graph of Φ. This is done as shown in the routine *precompute_token_flow* of Fig. 17(a). Given a graph G, and a complete configuration C for G, the routine first computes the interface token flow of every hierarchical actor in G by recursively calling itself. Then, G is set up as an SDF graph by resolving all the unknown quantities in the graph, using the complete configuration C. The process of configuration of G into an SDF graph, in the routine

configure_as_SDF is further elaborated in Fig. 17(b). This is followed by computation of the repetitions vector of G, after which the interface token flow is obtained easily. Note that the interface token flow determined in this fashion is a speculative quantity (based in general on default values of unknown parameters), and will be compared later with the actual interface token flow computed with real (instead of default) values for all the subinit graph parameters that are bound to subsystem dataflow inputs, and all the body graph parameters that are configured in the subinit graph. This entire process of computing the unknown token flow at the interface ports of the subinit and body graph of an hierarchical actor H can be interpreted as determining the port consumption and production functions κ_H and φ_H, and deriving the parameter interpretation function f_H.

Once all of the hierarchical actors have been processed in this fashion, the graph G is configured into an SDF graph by resolving the remaining unknown quantities in the graph (token flow, edge delays) to non-negative integer values as

```
function configure_graph(graph G, configuration C)
    configuration C_new = ∅
    foreach (p ∈ params(G))
        if ({p, *} ∈ C) C_new = C_new ∪ {p, C(p)}
        else
            if (default(p) = ⊥) error("default value must be specified")
            C_new = C_new ∪ {p, default(p)}
        end if
    end for
    return C_new
end function
```

Figure 16. The routine *configure_graph* that appears in the operational semantics of PSDF.

```
function precompute_interface_token_flow(graph G, configuration C)
    foreach hierarchical actor H in G
        Φ = subsystem(H)
        C_{Φ_s} = configure_graph(Φ_s, C)
        C_{Φ_b} = configure_graph(Φ_b, C)
        precompute_interface_token_flow(Φ_s, C_{Φ_s})
        precompute_interface_token_flow(Φ_b, C_{Φ_b})
    end for
    configure_as_SDF(G, C)
    compute_repetitions_vector(G)
    assign_interface_token_flow(G)
end function
```

(a)

```
function configure_as_SDF(graph G, configuration C)
    foreach leaf actor A in G
        C_A = config_{A, C}
        foreach port θ of A, assign_token_flow(θ, f_A(θ, C_A))
    end for
    foreach edge e in G
        C_e = config_{e, C}
        assign_delay(e, f_e(C_e))
    end for
end function
```

(b)

Figure 17. Two subroutines used in the operational semantics of PSDF — *precompute_interface_token_flow*, shown in (a), and *configure_as_SDF*, shown in (b).

shown in the routine *configure_as_SDF* in Fig. 17(b). For every leaf actor A, a complete configuration for the parameters of A is determined by augmenting the static configuration of A ($config_A$) with the configuration C of G, as specified in (26). This complete configuration is then used to resolve the unknown token flow at each port of actor A by calling the parameter interpretation function of A. As presented here, the parameter interpretation function of an actor is assumed to accept a complete configuration for the actor parameters and an actor port as inputs, and return the token flow at that port of the actor. This provides a more streamlined representation of the parameter interpretation function of an actor from that defined in Section 3.2. Similarly, the unknown delay characteristics of each edge is obtained as an integer by calling the parameter interpretation function of that edge with the complete parameter configuration of the edge. After configuring G as an SDF graph, the token transfer and max delay value bounds are verified for each actor and each edge respectively in G. A schedule is then determined for G by computing the repetitions vector and then constructing a valid schedule. Sample rate consistency and the existence of valid schedules are verified in this process. If G is an init graph or a subinit graph of a subsystem, then the init condition for local synchrony, or the subinit output condition for local synchrony (each actor configuring a parameter should be invoked once, producing one token at each interface output port) is also verified after computing the repetitions vector. Once a valid schedule has been constructed for G, actors are fired in the order specified by the schedule. Prior to executing the schedule, variables C_{internal} and C_{out} are initialized as empty graph configurations.

Firing a leaf actor L is equivalent to executing the (code of the) actor. If L configures a graph parameter at one of its output ports θ, then one of two actions is taken depending on the status of the output port θ. If θ is an interface output port of G, then the configuration C_{out} is augmented, and this will be returned later as the output of this routine. If however, θ is not an interface port, but an internal port of G, then it implies that the parameter configured in this fashion appears as a subinit graph parameter of one of the hierarchical actors in G, that represents subsystem Φ, and the parameter is bound to an interface input of Φ ($inputs_p(\Phi)$). In such cases, the configuration C_{internal} is augmented. If C_{internal} already contains a value for this parameter from a previous firing of L, then that entry is removed, and the new value is added to the configuration. While augmenting C_{out}, checking for previous values of the parameter is not necessary as for a locally synchronous specification, actors configuring parameter values at interface output ports are guaranteed to fire only once.

The process of firing a hierarchical actor L is broken up into five steps. In the first step, a configuration is assigned to the parameters of the subinit graph of the subsystem Φ that is represented by L. This configuration is assembled from the configuration of G (C), the configuration determined by the init graph of Φ ($C_{\Phi_{\text{init-set}}}$) and the configuration determined by parameters set at internal ports of G (C_{internal}). Note that these three configurations are a superset of all the subinit graph parameters, and hence, in assigning a configuration to Φ_s, default values of parameters will not be necessary. The second step is executing the subinit graph by recur-

sively calling the routine *execute* on the subinit graph, with the complete configuration of Φ_s. The parameters configured by the subinit graph at its output ports are returned in the configuration $C_{\Phi_{\text{subnit-set}}}$. The third and fourth steps consist of assigning a configuration to the body graph, assembled from $C_{\Phi_{\text{init-set}}}$ and $C_{\Phi_{\text{subnit-set}}}$ (in this case also, all parameters of the body graph are contained in these two configurations), and executing the body graph. The final step is that of verifying the subinit input condition and the body condition for local synchrony of Φ. After executing the subinit and body graphs in the previous steps, the repetition count of each interface actor is known exactly, and hence the interface token flow of H is fully specified. This represents the actual interface token flow of H, and is compared to the pre-computed interface token flow of H that was obtained earlier by performing speculative computation with default values of those subinit graph parameters that are bound to dataflow inputs of H, and those body graph parameters that are set in the subinit graph. In case of any mismatch between the actual and pre-computed interface token flow, a local synchrony error is flagged off for subsystem H. After the schedule for G is exhausted by firing each actor invocation in the schedule in this fashion, one invocation of graph G is completed, and the configuration C_{out} is returned as the configuration of the parameters set up at interface output ports of G.

Since, the topmost specification Φ embedded in the top-level graph G will not have any unspecified parameters, so execution of Φ is initiated by calling the routine *execute* on G with an empty configuration C. Since G consists of a single

hierarchical actor representing Φ, it does not have any interface output ports, and hence it returns the empty configuration C_{out}, that can be simply discarded.

We have seen that the PSDF operational semantics dictates that the subinit input condition and the body condition for local synchrony of subsystem Φ are verified by pre-computation of the interface token flow of Φ_s and Φ_b using default values of parameters of Φ_s that are bound to dataflow inputs of Φ, and parameters of Φ_b that are set in Φ_s, and subsequently comparing these with the actual interface token flow. Thus, in assigning default values to parameters, the user has to make judicious choices. In particular, only those parameters of Φ_s that are not bound to dataflow inputs of Φ, and those parameters of Φ_b that are not set up in Φ_s can have an unspecified ($\perp$) default value. All other parameters must have a specified default value such that their inter-relationship in determining the interface token flow of Φ is the same as any combination of values that the parameters can take on at run-time.

3.8. Consistency and Verification

Before discussing analysis and verification issues for PSDF, we first review some pre-requisite consistency notions for general DSP dataflow specifications, adapted from [7].

3.8.1 Binary Consistency and Decidable Dataflow

In general DSP dataflow specifications, the term consistency refers to the two essential requirements — the absence of deadlock and unbounded data accumu-

lation. An *inherently consistent* dataflow specification is one that can be implemented without any chance of buffer underflow (deadlock) or unbounded data accumulation (regardless of the input sequences that are applied to the system). If there exist one or more sets of input sequences for which deadlock and unbounded buffering are avoided, and there also exist one or more sets for which deadlock or unbounded buffering results, a specification is termed *partially consistent*. A dataflow specification that is neither consistent nor partially consistent is called *inherently inconsistent* (or simply *inconsistent*). More elaborate forms of consistency based on a probabilistic interpretation of token flow are explored in [23].

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

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

3.8.2 Consistency of PSDF Specifications

In PSDF, consistency considerations go beyond deadlock and buffer overflow. In particular the concept of consistency in PSDF includes local synchrony issues. As we have seen in Section 3.6, local synchrony consistency is, in general, dependent on the input sequences that are applied to the given system. Thus, it is clear that PSDF cannot be classified as a binary-consistency model. Furthermore, consistency verification for PSDF is not a decidable problem. In general, if a PSDF system completes successfully for a certain input sequence, the system may be inherently consistent, or it may be partially consistent. Similarly, if a PSDF system encounters a local synchrony violation for certain input sequences, the system may be inconsistent or partially consistent.

The elegance of PSDF lies in its robust operational semantics that accommodates, but does not rely on rigorous verification. There exists a well-defined concept of "well-behaved" operation of a PSDF specification, and the boundary between well-behaved and ill-behaved operation is also clearly defined, and can be detected immediately at run-time in an efficient fashion. More specifically, our development of parameterized dataflow provides a consistency framework and operational semantics that leads to precise and general run-time (or "simulation time") consistency verification. In particular, an inconsistent system (a specification together with an input set) in PSDF (or any parameterized version of one of the existing binary consistency models) will eventually be detected as being inconsistent, which is an improvement in the level of predictability over other models that go beyond binary

consistency, such as BDF, DDF, BDDF, and CDDF. In these alternative "dynamic" models, there is no clear-cut semantic criterion on which the run-time environment terminates for an ill-behaved system — termination may be triggered when the buffers on an edge are full, but this is an implementation-dependent criterion. Conversely, in PSDF, when the run-time environment forces termination of an ill-behaved system, it is based on a very well-defined semantic criterion that the system is either inconsistent or partially consistent.

In addition, implementation of the PSDF operational semantics can be streamlined by careful compile-time analysis. Indeed the PSDF model provides a promising framework for productive compile time analysis that warrants further investigation. As one example of such streamlining, we develop an efficient quasi-static scheduling algorithm in Sections 4.3. In our quasi-static scheduling framework, it is possible to perform symbolic computation, and obtain a symbolic repetitions vector of a PSDF graph, similar to what is done in BDF and CDDF. Then depending on how much the compiler knows about the properties of the specification through user assertions, some amount of analysis can be performed on local synchrony consistency. As implied by the operational semantics — which strictly enforces local synchrony — consistency issues that cannot be resolved at compile time must be addressed with run-time verification. Illustrations of consistency analysis using the symbolic repetitions vector technique can be found in Section 4.3.2.

The general verification problem for PSDF specifications is clearly non-trivial, and deriving effective, efficient verification techniques appears to be a promis-

ing area for further research. In particular, the issue of local synchrony verification of a PSDF subsystem calls for more investigation, as it arises as an exclusively PSDF-specific consideration that is inherent in the parameterized hierarchical structure that PSDF proposes. On the other hand, dataflow consistency issues (sample rate consistency and the presence of sufficient delays) are a by-product of the underlying SDF model, and have been explored before in a dynamic context in models such as BDF, CDDF, and BDDF.

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

every edge rules out unbounded token accumulation on an edge across invocations of a PSDF graph. A suitable bound for a PSDF graph $G = (V, E)$ can be expressed as

$$\max_{\bar{p} \in \overline{domain(G)}} \left(\sum_{\theta \in OUT(V)} [(\mathbf{q}_{G,\bar{p}}(actor(\theta))\varphi_{actor(\theta)}(\theta, config_{actor(\theta),\bar{p}}))] + \sum_{e \in G} \delta_e(config_{e,\bar{p}}) \right) , \quad (55)$$

which is

$$\leq \max_{\bar{p} \in \overline{domain(G)}} \left(\sum_{\theta \in OUT(V)} [(\mathbf{q}_{G,\bar{p}}(actor(\theta))\tau_{actor(\theta)}(\theta))] + \sum_{e \in G} \mu_e \right) . \quad (56)$$

We assume in (55), and (56) that $\mathbf{q}_{G,\bar{p}}$ exists for every $\bar{p} \in \overline{domain(G)}$. The token production and consumption quantities are bounded (by the max token transfer function), and the delay on an edge is also bounded (by the max delay value), as shown in (56). Since the token transfer at each actor port is bounded, there is only a finite number of possible different values that the repetitions vector can take on. Hence the maximum in (55) exists. Computing much tighter bounds may in general be possible, and this appears to be an useful direction for future work that warrants further investigation.

3.9. Actor Parameters and Dataflow Inputs

After having formally defined the PSDF model (Section 3.1 to 3.7), we would like to revisit the distinction between dataflow inputs and parameters of a PSDF actor. As discussed in Section 2.1, the value of a dataflow input can change across every invocation of the associated actor, and hence can be used to control the behavior of the actor at the granularity of every actor invocation. On the other hand, a parameter, in general, is configured once and maintains a constant value for a sequence of successive invocations of the actor. Thus parameters control actor behavior at a coarser level of granularity than dataflow inputs.

However, the invocation semantics of PSDF allows controlling an actor parameter through a dataflow input, thus allowing actor behavior to be controlled at the granularity of every actor invocation. An example is shown in Fig. 18. Fig. 18(a) shows a PSDF actor A with a single parameter *param1* and one input and output port. Each actor port is marked with the name of that port, and actor parameters are indicated within parentheses inside the actor. Typically, in an application scenario, *param1* will be so configured that in general, it will maintain a constant value over a number of successive invocations of A. However, there may be applications which demand that the parameter *param1* changes across every invocation of A, and take on values based on dataflow input from another actor, say B. One possible solution is shown in Fig. 18(b). *param1* is no longer modeled as an actor parameter of A, instead, an extra input port has been added to A where it accepts dataflow from B, and the value of the token at this input port is used internally to replace the function-

ality performed by *param1*. A different solution that utilizes the PSDF subsystem semantics is shown in Fig. 18(c). Here, the library specification of actor A remains unchanged, instead actor A is encapsulated inside the body graph of a new subsystem *S1*, and actor B provides dataflow input to the *Propagate* actor in the subinit graph of *S1*, and the unspecified actor parameter *param1* is configured at the output port of the *Propagate* actor. The functionality of the *Propagate* actor is simply to copy its input token to its output port. According to the PSDF invocation semantics, every invocation of A will be preceded by an invocation of *Propagate*, which will configure the parameter *param1* with the dataflow output of actor B, thus effectively allowing the parameter to be controlled by dataflow input.

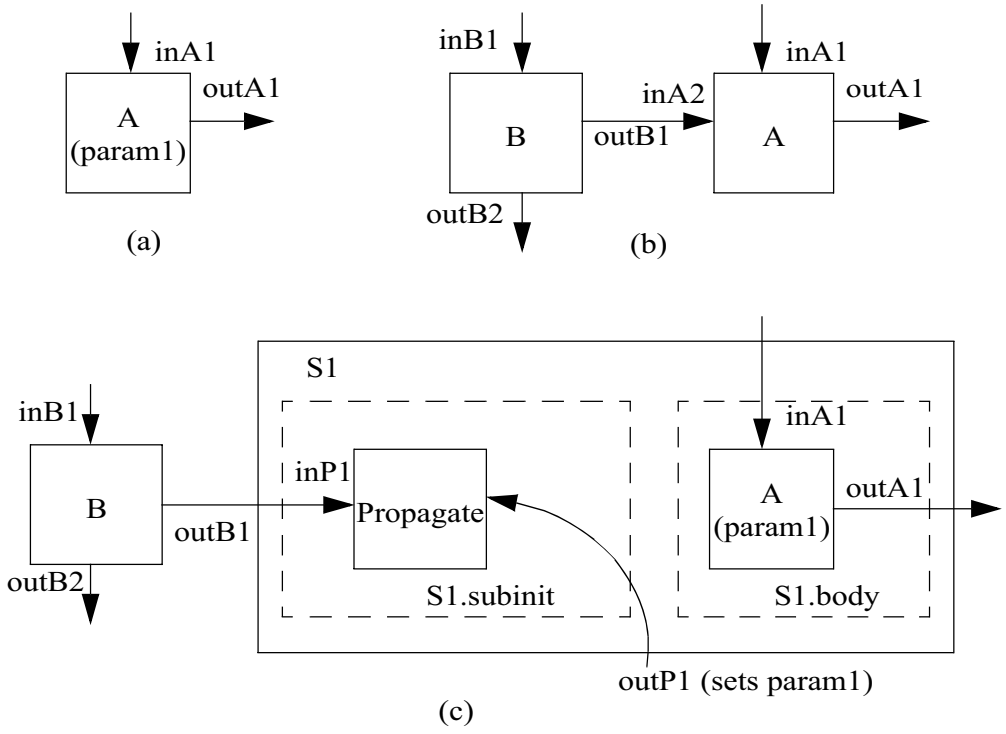


Figure 18. An example to demonstrate that actor parameters are strictly more general than dataflow inputs.

The above example demonstrates that in the parameterized framework, it is possible to simulate the functionality performed by a dataflow input through a parameter of the associated actor, which makes parameters strictly more general than dataflow inputs. This translates to increased design flexibility and modularity, and a condensed actor library in many cases, for block diagram DSP programming environments. For example, with a library specification of actor A as in Fig. 18(a), the application designer can fulfil various application-specific needs by suitably configuring the parameter *param1*, which includes assigning a fixed, static value to *param1* to be maintained over all invocations of A ; holding *param1* constant over a certain number of invocations of A , and allowing it to change across this window; and allowing *param1* to change across every invocation of A . This flexibility implies that it is not necessary to increase the size of the actor library by adding a different "version" of actor A , as in Fig. 18(b).

3.10. The PSDF Application Model and the Formal Model

As the reader may have noticed, the intuitive, informal PSDF model presented in Chapter 2 is somewhat different from the formal model developed in Sections 3.1 to 3.7. Henceforth we refer to the former as the "Application Model", and to the latter as the "Formal Model". In this section, we explain why we have preserved both models, and how we can map from one model to the other.

In the formal model, a bottom up approach is adopted where unspecified actor or edge parameters propagate "upwards" as graph parameters, and then as specification parameters. The init and subinit graphs are responsible for configuring

these unspecified parameters. In the application model, we have a top-down approach, where subsystem parameters are specified separately and propagate "downwards", with unspecified actor and edge parameters being configured with appropriate subsystem parameters. In this case, the init and subinit graphs configure the subsystem parameters. By *specification parameter* we refer to the definition of the formal model, while by *subsystem parameter* we refer to the definition in the application model. Whenever we say "parameter of a subsystem", it will always be clear from context whether we are referring to a specification parameter, or to a subsystem parameter.

There isn't any fundamental conceptual difference between the application and formal models, except for technical details. However, each has justified reasons for a separate existence. From our experience with designing applications in PSDF, we felt that the application model provides a more natural, intuitive style from the application designer's perspective. Typically the algorithm for the application will have some natural parameters that can be modeled as subsystem parameters. At the top level, the functionality can be broken up into separate logical units (subsystems) again with its own natural set of parameters, and so on. While designing practical systems, actors will be selected from an actor library, and the user will configure each actor according to the application. In SDF systems, actor parameters are assigned fixed integer values. As a natural extension to this concept, PSDF also allows actor parameters to be assigned subsystem parameter values. Accordingly, all the PSDF examples and applications in this report have been developed in the appli-

cation model. On the other hand, the formal model is useful for precise specification and analysis of the operational semantics, consistency issues, and synthesis tasks. Actor parameters have a well defined notion of a domain set, and allowing actor parameters to be assigned arbitrary subsystem parameter values needlessly clutters the notion of this domain set.

In Fig. 19, we sketch an algorithm for mapping a PSDF specification Φ specified in the formal model to a specification in the application model. The reverse mapping of a PSDF specification Φ from the application model to the formal model is shown in Fig. 20 and Fig. 21.

From the construction specified in Fig. 19, 20, and 21, we see that the subsystem internal parameters in the application model are equivalent to the graph

Step 1: Process the init graph Φ_i: For each interface output port $\theta \in \mathit{outputs}(\Phi_i)$ in the init graph Φ_i, assign an init-configured external subsystem parameter for Φ called p_θ, to be set from that port. In the subinit graph Φ_s, insert assignment statements of the form $\Phi_{ts}(\theta) = p_\theta$, for each $\theta \in \mathit{ToSubinit}(\Phi_i)$, which configures each actor/edge parameter of the subinit graph with the corresponding subsystem parameter. In the body graph, insert assignment statements $\Phi_{tb}(\theta) = p_\theta$ for each $\theta \in \mathit{ToBody}(\Phi_i)$.
Step 2: Process the subinit graph Φ_s: For each interface output port $\theta \in \mathit{outputs}(\Phi_s)$ in the subinit graph Φ_s, assign an internal subsystem parameter for Φ called p_θ, to be set from that port. In the body graph, insert assignment statements $\Phi_{fs}(\theta) = p_\theta$. For each interface input port $\theta \in \mathit{inputs}_p(\Phi)$ of the specification Φ that is bound to a parameter in the subinit graph, assign a parent-configured external subsystem parameter p_θ, and bind it to the port θ. In the subinit graph, insert assignment statements of the form $\Phi_{ps}(\theta) = p_\theta$.
Step 3: For each hierarchical actor H in the specification Φ, go to Step 1 with $\Phi = \mathit{subsystem}(H)$.

Figure 19. An algorithm to map a PSDF specification in the PSDF Formal Model to an equivalent specification in the PSDF Application Model.

parameters $\Phi_{\text{fs}}(\mathit{outputs}(\Phi_s))$ in the formal model. Similarly there is an equivalence between the parent-configured subsystem parameters of the application model and the graph parameters $\Phi_{\text{ps}}(\mathit{inputs}_p(\Phi))$ in the formal model. Thus the subinit input condition for local synchrony of a subsystem (49, 51) can be interpreted as the requirement that *the number of tokens consumed at each interface input port of the subinit graph should be invariant over the parent-configured subsystem parameters of the associated subsystem*. Similarly, the body condition for subsystem local synchrony (52, 53, 54) can be restated as the requirement that *the token flow at each interface port of the body graph should be invariant over the internal subsystem parameters of the associated subsystem*. In the subsequent chapters, we will usually

Step 1: For each hierarchical actor H in Φ, process that child subsystem first (i.e. go to Step 1 with $\Phi = \mathit{subsystem}(H)$), and then proceed to Step 2.
Step 2: Process the body graph Φ_b: Start with an empty set of graph parameters $\mathit{params}(\Phi_b)$, and empty functions Φ_{tb} and Φ_{fs}. Consider each assignment statement of the form $a_p = p_\theta$ that configures an actor parameter or an edge parameter a_p with an immediate subsystem parameter p_θ, where θ gives the output port of the init or subinit graph that configures the subsystem parameter p_θ. Since the body graph can only use immediate subsystem parameters, mark each such actor parameter as a graph parameter, i.e. $\mathit{params}(\Phi_b) = \mathit{params}(\Phi_b) \cup a_p$. If $\theta \in \mathit{outputs}(\Phi_i)$ (the subsystem parameter is configured from the init graph), then insert the element (θ, a_p) into the function Φ_{tb}, which is equivalent to introducing the mapping $\Phi_{\text{tb}}(\theta) = a_p$. Otherwise, if $\theta \in \mathit{outputs}(\Phi_s)$ (the subsystem parameter is configured from the subinit graph), then insert the element (θ, a_p) into the function Φ_{fs}. If more than one actor parameter is configured with the same subsystem parameter, then direct an edge from port θ to a fork actor with as many output ports as the number of actor parameters that map to the same subsystem parameter. Mark each output port of the fork actor as an interface output port of the graph, and insert the corresponding mappings for each interface port.

Figure 20. The first two steps of an algorithm to map a PSDF specification in the PSDF Application Model to an equivalent specification in the PSDF Formal Model.

Step 3: Process the subinit graph Φ_s: Start with an empty set of graph parameters $params(\Phi_s)$, an empty set of specification parameters $params(\Phi)$, and empty functions Φ_{ts} and Φ_{ps}. Consider each assignment statement of the form $a_p = p_\theta$ that configures an actor parameter or an edge parameter a_p with an immediate subsystem parameter p_θ. Here, θ denotes the output port of the init graph that configures this parameter, or the input port of the subsystem to which this parameter is bound. Mark each such actor parameter as a graph parameter — $params(\Phi_s) = params(\Phi_s) \cup a_p$. If $\theta \in outputs(\Phi_i)$ (the subsystem parameter is configured from the init graph), then insert the member (θ, a_p) into the function Φ_{ts}. Otherwise, if $\theta \in inputs_p(\Phi)$ (the parameter is bound to an input port of the subsystem), then insert the element (θ, a_p) into the function Φ_{ps}. After immediate subsystem parameters, consider those actor/edge parameters that are configured with inherited subsystem parameters, for which $a_p = p$, where p is an inherited subsystem parameter. Mark each such actor parameter as a graph parameter, and also as a specification parameter — $params(\Phi_s) = params(\Phi_s) \cup a_p$, and $params(\Phi) = params(\Phi) \cup a_p$. In addition, insert the assignment statement $a_p = p$ in the parent graph of the subsystem, which will ensure that these specification parameters are dealt with in the parent graph.

Step 4: Process the init graph Φ_i: Start with an empty set of graph parameters $params(\Phi_i)$. Consider each assignment statement of the form $a_p = p$ that configures an actor parameter a_p with the inherited subsystem parameter p. Recall that the init graph can use only inherited subsystem parameters. Mark each such actor parameter as a graph parameter and as a specification parameter: $params(\Phi_i) = params(\Phi_i) \cup a_p$ and $params(\Phi) = params(\Phi) \cup a_p$. Also, insert the assignment statement $a_p = p$ in the parent graph of the subsystem.

Figure 21. The third and fourth steps of an algorithm to map a PSDF specification in the PSDF Application Model to an equivalent specification in the PSDF Formal Model.

use the notation for subsystem parameters, and utilize this interpretation of the sub-init input condition and the body condition for local synchrony in explaining local synchrony verification techniques.

Chapter 4. Scheduling PSDF Specifications

In this chapter, we develop scheduling techniques for PSDF specifications, including quasi-static scheduling techniques and run-time scheduling techniques.

4.1. Parameterized Looped Schedules

First, we define some quasi-static scheduling terminology for PSDF. This terminology is based on the framework of looped schedules developed for SDF graphs [5, 8]. In this section and in the next sections (Section 4.2, 4.3) we will use the simplified PSDF graph as the basis of our scheduling techniques, and accordingly, we will use the corresponding notation, as described in Section 3.4.1.

4.1.1 Looped Schedules, Clustering, and APGAN in SDF

Given an SDF graph G, a *schedule loop* is a parenthesized term of the form $(nT_1T_2\ldots T_m)$, where n is a positive integer, and each T_i is either an actor in G or another schedule loop. The parenthesized term $(nT_1T_2\ldots T_m)$ represents the successive repetition n times of the invocation sequence $T_1T_2\ldots T_m$. If $L = (nT_1T_2\ldots T_m)$ is a schedule loop, then n represents the *iteration count* of L, each T_i is an *iterand* of L, and $T_1T_2\ldots T_m$ constitutes the *body* of L. A *looped schedul*e is a sequence $V_1V_2\ldots V_k$, where each V_i is either an actor or a schedule loop. Given a looped schedule S, it is called a *single appearance schedule* if each actor in G appears only once in the schedule.

Given a connected, consistent SDF graph $G = (V, E)$, and a subset of

actors $Z \subseteq V$, the *repetition count* of Z is defined as

$$q_G(Z) \equiv gcd(\{\mathbf{q}_G(A) | (A \in Z)\}), \tag{57}$$

and can be viewed as the number of times a minimal periodic schedule for the subset of actors in Z is invoked in G. If A and B are adjacent actors, then their repetition count is denoted as

$$\rho_G(\{A, B\}) \equiv q_G(\{A, B\}). \tag{58}$$

Given an actor $\Omega \notin V$, *clustering* Z into Ω means generating the new SDF graph $(\bar{V}, \bar{E})$, such that

$$\bar{V} = V - Z + \{\Omega\}, \tag{59}$$

and

$$\bar{E} = E - (\{e | (src(e) \in Z) \mathbf{or} (snk(e) \in Z)\}) + E^*, \tag{60}$$

where E^* is a "modification" of the set of edges that connect actors in Z to actors outside of Z. If for each $e \in E$ such that $src(e) \in Z$ and $snk(e) \notin Z$, $\bar{e}$ is defined by

$$src(\bar{e}) = \Omega,\ snk(\bar{e}) = snk(e),$$

$$d(\bar{e}) = d(e),\ p(\bar{e}) = p(e) \times ((\mathbf{q}_G(src(e))) / (q_G(Z))), \text{ and } c(\bar{e}) = c(e)$$

and similarly, for each $e \in E$ such that $snk(e) \in Z$ and $src(e) \notin Z$, $\bar{e}$ is defined by

$$src(\bar{e}) = src(e),\ snk(\bar{e}) = \Omega,$$

$d(\bar{e}) = d(e)$, $p(\bar{e}) = p(e)$, and $c(\bar{e}) = c(e) \times ((\mathbf{q}_G(snk(e)))/(q_G(Z)))$

then, E^* can be specified by

$$E^* = \{\bar{e} | ((src(e) \in Z) \textbf{and} (snk(e) \notin Z)) \textbf{or} ((snk(e) \in Z) \textbf{and} (src(e) \notin Z))\} \quad . \tag{61}$$

The graph that results from clustering Z into Ω in G is denoted by $cluster(Z, G, \Omega)$ or simply $cluster(Z, G)$. Intuitively, an invocation of Ω in $cluster(Z, G)$ corresponds to an invocation of a minimal valid schedule for the sub-graph formed by the actors of Z in G. Z is *clusterable* if $cluster(Z, G)$ is consistent, and if G is acyclic, then Z *introduces a cycle* if $cluster(Z, G)$ contains one or more cycles. Fig. 22 shows an example of clustering in SDF graphs, given in [8]. Fig. 22(b) represents $cluster(\{B, C\}, G, \Omega)$, where G denotes the SDF graph in Fig. 22(a). For the example shown in the figure, $\mathbf{q}_G(A, B, C, D) = (3, 30, 20, 2)$, and thus $\rho_G(\{B, C\}) = 10$.

APGAN (Acyclic Pairwise Grouping of Adjacent Nodes) is a scheduling technique developed for acyclic SDF graphs geared towards joint code and data

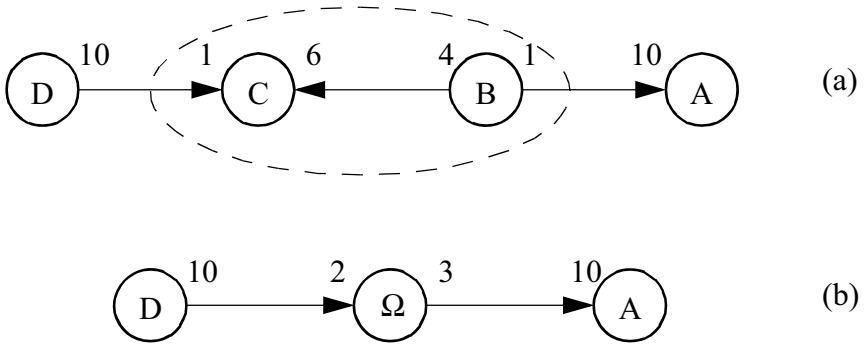


Figure 22. An example of clustering in SDF.

minimization objectives. Given a consistent, acyclic SDF graph as input, APGAN produces a minimal periodic, single appearance looped schedule that is shown to be optimal for a certain class of SDF graphs. In the APGAN technique, a cluster hierarchy is constructed by clustering exactly two adjacent vertices at each step. At each clustering step, an adjacent pair is chosen as an *APGAN candidate*, if clustering it does not introduce a cycle, and its repetition count is greater than or equal to all other adjacent pairs that do not introduce cycles. After the cluster hierarchy is constructed, APGAN outputs a schedule corresponding to the recursive traversal of the cluster hierarchy.

It is straightforward to construct a single-appearance schedule for the subgraph G_i corresponding to cluster Ω_i. Each such subgraph consists of only two actors X_i and Y_i, such that all edges in G_i are directed from X_i to Y_i. Depending on the value of $d(e)$ for each edge e, the optimal schedule is either

$$(\mathbf{q}_{G_i}(Y_i)Y_i)(\mathbf{q}_{G_i}(X_i)X_i), \tag{62}$$

or

$$(\mathbf{q}_{G_i}(X_i)X_i)(\mathbf{q}_{G_i}(Y_i)Y_i). \tag{63}$$

Starting with a schedule for the top-level subgraph, APGAN recursively goes down one level of hierarchy into the subgraph corresponding to a child cluster, and the flattened schedule of this child subgraph replaces its corresponding hierarchical actor in the top-level schedule.

4.1.2 PSDF Looped Schedules

Given a PSDF graph G, a *parameterized schedule loop* L represents successive repetition of an invocation sequence $T_1 T_2 \ldots T_m$, where each T_i is either a leaf actor in G, or a hierarchical actor in G, or another parameterized schedule loop. However, unlike SDF, the iteration count of the schedule loop, denoted as $loopcnt(L)$ is no longer an integer, but is a symbolic expression consisting of constants, subsystem parameters of the parent specification of G, init-configured subsystem parameters of child specifications represented by hierarchical actors present in the invocation sequence $T_1 T_2 \ldots T_m$, and compiler-generated variables. If T_i is a hierarchical actor in G representing specification Φ, then in order to compute a schedule for T_i, the (interface) token flow of T_i is necessary. This is obtained by computing a parameterized looped schedule (defined shortly) for Φ_s and Φ_b, and evaluating the token flow at the interface ports of Φ_s and Φ_b with default values of non-init-configured subsystem parameters of Φ. An occurrence of T_i in L can be replaced by $(VLS_s)(S_{\Phi_s})(VLS_b)(S_{\Phi_b})$, where S_{Φ_s} and S_{Φ_b} represent parameterized looped schedules for the subinit graph of Φ and the body graph of Φ respectively, and VLS_s and VLS_b consists of code that checks, respectively, the subinit input condition and the body condition for local synchrony of Φ. The code in VLS_s (VLS_b) consists of conditionals that compare the actual token flow at the interface input (input and output) ports of the subinit (body) graph (evaluated with the actual values of the non-init-configured subsystem parameters of Φ) with the pre-computed interface token flow (evaluated with the default values of non-init-configured

subsystem parameter), and flag off local synchrony errors in case of any mismatches. These code is together referred to as the *synchrony check* code of Φ.

Suppose that G is a PSDF graph that contains m hierarchical actors $H_1, H_2, \ldots, H_m$. A *parameterized looped schedule* S_G for G consists of three parts:

$$S_G \equiv (S_1, S_2, \ldots, S_m)(preamble_S)(body_S), \tag{64}$$

where each S_i represents a parameterized looped schedule for the init graph of $subsystem(H_i)$. Thus, the first part of the schedule for G, called the *initChild* phase of S, consists of successive invocations of the parameterized looped schedules of the init graphs of the child subsystems of G. The third part of the schedule $body_S$, called the *body* of S, is a sequence $V_1 V_2 \ldots V_k$, where each V_i is either an actor (leaf or hierarchical) in G or a parameterized schedule loop. The second part of the schedule $preamble_S$, called — the *preamble* of S, consists of code that configures the iteration count of each schedule loop in $body_S$ by defining in a proper order every compiler-generated variable used in the symbolic expression of that iteration count, and includes conditionals for checking sample rate consistency, and bounded memory consistency of graph G. Additionally, if G is an init graph, $preamble_S$ includes conditionals for checking the init condition for local synchrony of the parent specification of G (each actor in the init graph that configures a parameter value at an output port must produce exactly one token at that port). Similarly, if G is a

subinit graph, then $preamble_S$ includes conditionals for checking the subinit output condition for local synchrony of the parent specification of G.

A parameterized looped schedule S for a PSDF graph G is a *single-appearance schedule*, if each actor in G appears only once in $body_S$.

The concept of clustering in PSDF graphs is exactly similar to the SDF definition, except that in addition to integer arithmetic, we now also have symbolic computations being performed. We explain the PSDF clustering process, and the parameterized APGAN scheduling technique in Section 4.3.1, through examples.

4.2. Scheduling Strategy

The PSDF operational semantics allows streamlined implementation by careful compile-time analysis. However, to avoid dependence on any particular type of compile-time analysis or optimization, the operational semantics is defined in terms of a minimal set of requirements for correct, locally synchronous execution. Any implementation that guarantees these requirements is a valid implementation of the operational semantics. Our endeavor, in scheduling PSDF specifications, is to streamline the implementation of the operational semantics by generating a quasi-static schedule whenever possible, and otherwise fall back on a run-time scheduler that implements the operational semantics in a straightforward manner. At run-time, every PSDF graph assumes an SDF configuration on every invocation, and hence can be analyzed and scheduled using available SDF techniques. The run-time kernel can utilize quasi-static schedules determined at compile-time for sub-parts of the specification.

We perform quasi-static scheduling for all PSDF acyclic graphs, and for a certain class of cyclic graphs that we call *simple cyclic graphs*. In such cyclic graphs, each fundamental directed cycle has a single delay element with a known value, and each such fundamental cycle is statically known to be a single-rate system (i.e. $p(e) = c(e)$ for each edge e in the fundamental cycle), or can be configured into a single-rate system. Simple cyclic graphs arise frequently in practical systems that incorporate feedback loops, leading to graph cycles with a single feedback edge containing delay tokens.

For simple cyclic graphs, it is easy to determine if they are free of deadlock (i.e. whether sufficient numbers of delay tokens exist in every fundamental directed cycle in the graph) to enable construction of a periodic schedule [8]. In a deadlock-free simple cyclic graph, we can effectively "break" the cycles and convert the graph into an acyclic graph, for scheduling purposes. If the amount of delay on a feedback edge e' is less than the number of tokens consumed at $snk(e')$, then the graph is detected as being deadlocked. Otherwise, if edge e' has an associated delay that is at least equal to the numbers of tokens consumed by the corresponding sink actor ($snk(e')$), then we break the dependence associated with e', which is equivalent to removing e' from the cycle that it belongs to, for the purpose of constructing a periodic schedule for the subgraph comprising the associated cycle. This removal of e' is possible because in the associated cycle, the repetitions vector components are identically equal to unity [8], and thus $snk(e')$ does not depend on data produced by $src(e')$ within a given schedule period of the associated cycle. Such

breaking of dependencies in single-rate data dependence structures has been studied extensively in the context of vectorization (see for example [2]) for imperative programming languages. Systematically breaking cyclic dependencies in the context of general (possibly multi-rate) SDF graphs has been explored in [4].

Thus, in a fundamental directed cycle in a simple cyclic graph, if the feedback edge e' possesses sufficient delay tokens (at least equal to the numbers of tokens consumed by $snk(e')$), then $snk(e')$ does not depend on data produced by $src(e')$ in a given invocation of a (periodic) schedule of the fundamental cycle. However, the periodic schedule for the entire simple cyclic graph may in general, consist of several invocations of a periodic schedule of each fundamental cycle, and data dependencies implied by feedback edges (from $src(e')$ to $snk(e')$) in a fundamental cycle become relevant across invocations of their periodic schedules. Consequently, this data dependence has to be preserved in deriving a periodic schedule for the entire simple cyclic graph. Hence, the feedback edge e' cannot be simply removed from the graph, its presence has to be utilized for scheduling purposes, for example, in choosing adjacent vertices for clustering (Section 4.3.1).

To process simple cyclic graphs, the quasi-static scheduler is provided with a pre-processor which examines an input cyclic graph to check if it is eligible as a deadlock-free simple cyclic graph, and if so then break every cycle by removing the feedback edge; assign an unique identifier to each fundamental cycle; mark each actor with the identifier(s) of the fundamental cycle(s) that it belongs to; and provide the resulting marked acyclic graph for scheduling.

From our application design experience it appears that a large class of useful DSP applications falls under the categories of acyclic graphs and simple cyclic graphs. In fact, efficient, quasi-static schedules can be computed for all the examples and applications presented in this report.

4.3. Quasi-static Scheduling

In this section, we explain the quasi-static scheduling technique that we have developed for PSDF specifications. We develop an extension of APGAN to PSDF graphs, explain consistency analysis issues, present the complete quasi-static scheduling algorithm, make some observations on incorporating re-initializable delays into our quasi-static scheduling framework, and finally provide some examples of our quasi-static schedules.

4.3.1 Parameterized APGAN — P-APGAN

The basic step in quasi-static scheduling of a PSDF specification is to determine the body ($body_S$) of the parameterized looped schedule S for a PSDF graph G. We use an extension of the APGAN scheduling technique (Section 4.1.1) to derive minimal periodic, single-appearance parameterized schedule loops representing the body of the parameterized schedule for G. This extended APGAN scheduling technique is called *P-APGAN* for *parameterized APGAN*.

For a PSDF graph, it is not in general possible to select an adjacent pair of actors for clustering based on the maximum value of the repetition count of the cluster, as the repetition count can only be obtained symbolically. At present, given a

choice among adjacent pairs of actors such that clustering each adjacent pair does not introduce a cycle in the graph (we refer to each such adjacent actor pair as a *P-APGAN candidate*), we use the following scheme for selecting an adjacent pair for clustering. Recall that a category of cyclic graphs are presented for P-APGAN clustering after breaking the cycles in these graphs, and uniquely marking actors that belong to the same fundamental cycle. Two adjacent actors that belong to the same fundamental cycle in the original graph are assigned a higher priority for clustering compared to adjacent actors that do not satisfy this criterion. More precisely, if a P-APGAN candidate shares at least one common marking between the two actors that comprise the P-APGAN candidate, then it is assigned a higher priority for clustering compared to other P-APGAN candidates that do not share any common marking. Ties are broken arbitrarily. For example, given the following five choices of P-APGAN candidates in terms of actor markings — [(*s1*, *s2*), (*s1*)], [(*s1*, *s2*), *s2*)], [(*s2*), (*s1*)], [(*s1*), (NULL)], and [(NULL), (NULL)], where the notation [(*s1*, *s2*), (*s1*)] indicates a P-APGAN candidate with the first actor belonging to two fundamental cycle *s1* and *s2* of the original graph, and the second actor belonging to the fundamental cycle *s1*; a marking of NULL indicates that the actor does not belong to any fundamental cycle — the three P-APGAN candidates [(*s1*, *s2*), (*s1*)], [(*s1*, *s2*), *s2*)], and [(NULL), (NULL)] are assigned the highest priority for clustering, and any one among the three can be chosen. This selection scheme among P-APGAN candidates ensures that all actors belonging to the same fundamental cycle of the original graph are completely clustered among themselves and reduced to a single cluster,

before any clustering is performed with actors outside the fundamental cycle. Thus, all data precedences present in the original graph are always maintained in the quasi-static schedule. Among P-APGAN candidates with the same clustering priority, we are experimenting with giving priority to SDF edges (edges for which $p(e)$ and $c(e)$ are constant and known at compile time), and single-rate edges (edges for which it is statically known that $p(e)$ is equal to $c(e)$, but they are not necessarily known to be constant), with the goal of simultaneously minimizing code size and run-time computation in the quasi-static schedule. Selecting an adjacent pair for clustering such that both code size and data size are minimized, at least for a certain class of graphs (as is guaranteed by the original APGAN) is an interesting area for further research.

The schedule for the subgraph corresponding to each cluster is constructed by symbolic computation, and code is generated that does the actual computation at run-time. Clustering of two adjacent vertices in P-APGAN is shown in Fig. 23. Fig. 23(a) shows a portion of a PSDF graph G, represented by the subgraph associated with actors A and B. The symbols a, p, q, and b either represent subsystem parameters of the parent specification of G, or are compiler-generated variables representing unknown token flow, and will be suitably initialized with the parameter interpretation functions of the corresponding actors in the preamble code of the schedule for G. The two vertices are clustered into the single vertex Ω_1, and the topology of the graph, along with the token flow on the relevant edges is adjusted as shown in Fig. 23(b). Fig. 23(c) shows a pseudocode representation of the schedule

of the subgraph corresponding to cluster Ω_1, which is constructed as

$$((q/g)(A))((p/g)(B)) \text{ —} \tag{65}$$

(q/g) invocations of A, followed by (p/g) invocations of B, where g is a compiler-generated variable defined as $g = gcd(p, q)$. We say that $reps(A) = q/g$, and $reps(B) = p/g$, where $reps(A)$ denotes the *local repetition factor* of actor A, as opposed to the global repetition count $\mathbf{q}_G(A)$. The schedule in (65) for the subgraph $G_{A,B} = subgraph(\{A, B\})$ of Fig. 23(a) is obtained from the APGAN technique given in (62), (63). In particular note that $\mathbf{q}_{G_{A,B}}(A) = (q/g)$ and $\mathbf{q}_{G_{A,B}}(B) = (p/g)$, thus (63) leads to (65).

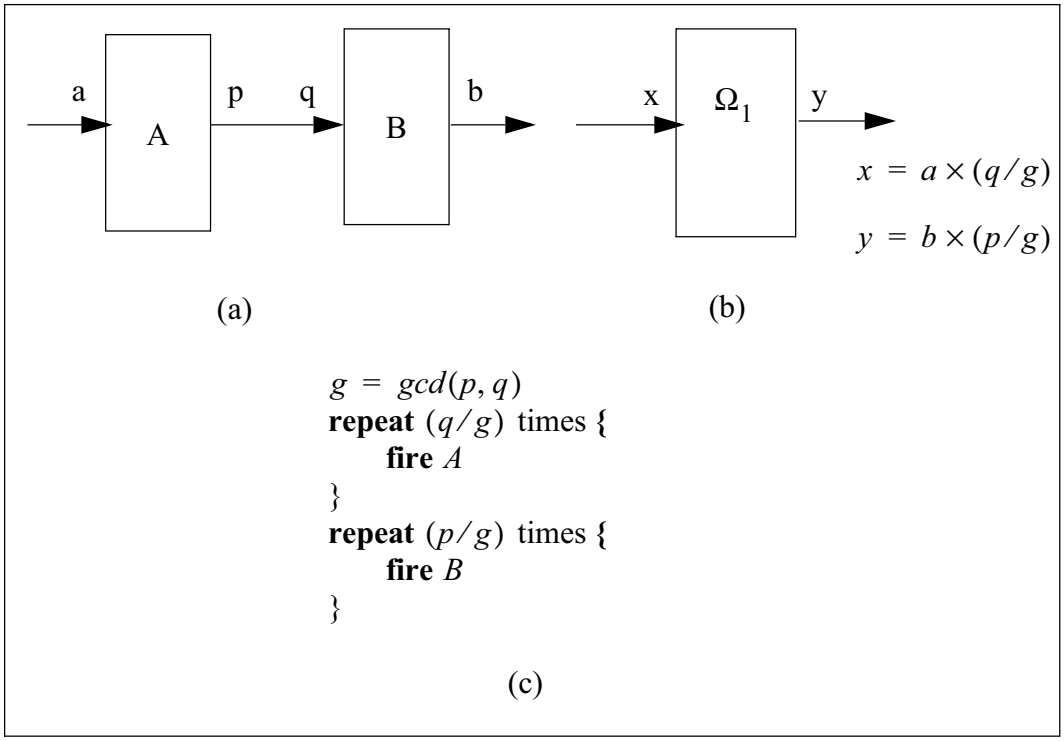


Figure 23. A clustering step in P-APGAN.

For each edge $\bar{e}$ going into actor A that does not belong to the subgraph associated with $\{A, B\}$, the token consumption on $\bar{e}$ is modified from $c(\bar{e})$ to $c(\bar{e}) \times reps(A)$, and the edge is re-directed to the cluster Ω_1 — $snk(\bar{e}) = \Omega_1$. For each edge $\bar{e}$ coming out of A that does not belong to the subgraph of A and B, the token production $p(\bar{e})$ and the source vertex $src(\bar{e})$ are modified in an analogous fashion. Edges incident to actor B are also processed similarly. This modification of the graph topology and the token transfer on the edges incident on $subgraph(\{A, B\})$ is derived from the clustering process for SDF graphs described in Section 4.1.1 in (59), (60), and (61). According to the latter, for an edge $\bar{e}$ going into actor A such that $\bar{e} \notin subgraph(\{A, B\})$, the tokens consumed from that edge should be modified to $c(\bar{e}) \times ((\mathbf{q}_G(A))/(q_G(\{A, B\})))$, after the clustering is completed. As stated above, our clustering process modifies the token consumption to $c(\bar{e}) \times reps(A)$. We show a derivation in Fig. 24 that the two are equivalent. The

$\mathbf{q}_G(A)/q_G(\{A, B\})$

$= \mathbf{q}_G(A)/(gcd(\mathbf{q}_G(A), \mathbf{q}_G(B)))$ [from (57)]

$= \mathbf{q}_G(A)/\left(gcd\left(\mathbf{q}_G(A), \left(\mathbf{q}_G(A) \times \frac{p(e)}{c(e)}\right)\right)\right)$ [from the balance equation for e]

$= (\mathbf{q}_G(A) \times c(e))/(gcd((\mathbf{q}_G(A) \times c(e)), (\mathbf{q}_G(A) \times p(e))))$ [multiplying numerator and denominator by $c(e)$]

$= (\mathbf{q}_G(A) \times c(e))/(\mathbf{q}_G(A) \times gcd(c((e), p(e))))$

$= (c(e))/(gcd(c((e), p(e))))$

$= reps(A)$

Figure 24. A derivation to demonstrate that the basic P-APGAN clustering step shown in Fig. 23 is equivalent to the clustering technique for SDF graphs described in (59) - (61).

derivation demonstrates that in the two-vertex cluster of Fig. 23(a), comprising actors A, B, and edge e, the repetition count of the source actor A divided by the repetition count of the subgraph associated with the two vertices $\{A, B\}$ is the same as the local repetition factor of A. A similar derivation can be made for the sink actor B of the two-vertex cluster of Fig. 23(a) to show that $\mathbf{q}_G(B)/q_G(\{A, B\})$ is the same as $reps(B)$.

Since the data size optimization objective has been dropped in going from APGAN to P-APGAN, a clustering step in P-APGAN becomes an entirely local computation, and does not need any global knowledge about the repetitions vector of the graph. For the same reason, in constructing the schedule of a subgraph corresponding to a cluster, P-APGAN does not take into account the delay on edges present in the subgraph, and the source actor in each two-vertex cluster always fires before the corresponding sink actor, as shown in Fig. 23, and (65).

While scheduling a nested PSDF graph G, interface edges connecting actors in G with actors in the parent graph G_p are "hidden" in G, but appear in G_p. An example is shown in Fig. 25. Part (a) of the figure shows a PSDF graph containing a single hierarchical actor H. Part (b) shows the body graph G of the PSDF subsystem corresponding to H. Actors B and C in G communicate with the parent graph actors A and D through the subsystem ports *port1* and *port2*, respectively. In the parent graph, G_p, the edge from A to H (at *port1*) and the edge from H (at *port2*) to D appear as normal edges in the graph. However, in G, the corresponding edges (from *port1* to B, and from C to *port2*) are indicated by dashed lines and are

hidden in the graph. In clustering the interface actors in G (actors B and C in this case), the P-APGAN algorithm is responsible for updating the connectivity and token flow of such hidden interface edges in accordance with the basic clustering step shown in Fig. 23, as if the interface edges are normal edges incident on the interface actors. More precisely, for an input interface edge e (the edge from *port1* to B), $c(e)$, and $snk(e)$ have to be updated as a part of a clustering step involving $snk(e)$, while for an output interface edge e (the edge from C to *port2*), $p(e)$ and $src(e)$ have to be updated as a part of a clustering step involving $src(e)$.

4.3.2 Consistency Analysis

Along with scheduling, the compiler has to analyze a PSDF specification for four types of consistency: sample rate consistency of a PSDF graph, bounded memory consistency of a PSDF graph, unit transfer consistency of a PSDF subsystem (the init and subinit graphs produce exactly one token on each output port at each invocation) and local synchrony consistency of a PSDF subsystem (the subinit input

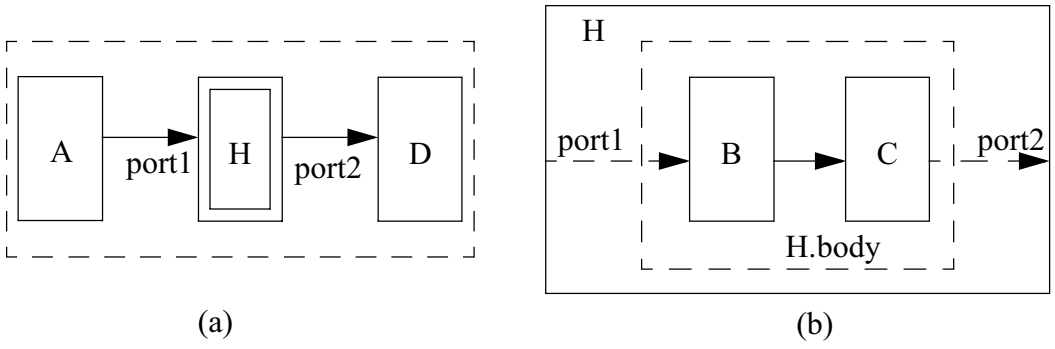


Figure 25. An example to demonstrate how interface edges connecting actors in a graph inside a subsystem to actors in the parent graph of the subsystem, affect P-APGAN clustering.

condition and the body condition for local synchrony). In general, for a graph containing one or more directed cycles, consistency also requires that sufficient delay is present in every directed cycle. For PSDF specifications however, we are not concerned with the latter, as our quasi-static scheduling techniques are limited to acyclic graphs, or cyclic graphs in which all cyclic dependencies are easily broken (as described in Section 4.2).

The general strategy followed for consistency analysis is to attempt to detect consistency compliance or violation at compile-time, and terminate execution in case of the latter. If the compiler does not have sufficient information to reach a definitive conclusion at compile-time, then code is generated to verify consistency at run-time, and terminate execution in case of a run-time consistency violation.

For sample rate consistency analysis, it is possible to compute the repetitions vector of the PSDF graph symbolically, as in the BDF or CDDF model, e.g. by the *depth-first-search* method [8]. Then the balance equations can be verified symbolically for each edge in the graph. For an edge e, with $c(e) = 3$, $p(e) = p$, if the repetitions vector has been computed as $\mathbf{q}(src(e)) = p$, $\mathbf{q}(snk(e)) = 3$, then the balance equation for edge e takes on the form $3p = 3p$, and it can be verified at compile-time that the balance equation for edge e is satisfied. If the balance equations are similarly satisfied for every edge in the graph, then the graph can be detected as sample-rate consistent at compile-time only. Otherwise, if the balance equation for at least one edge in the graph cannot be verified at compile time, then a run-time sample rate consistency check has to be performed. For example, if for

edge e the repetitions vector has been computed as $\mathbf{q}(src(e)) = x$, $\mathbf{q}(snk(e)) = y$, then the balance equation for edge e takes on the form $px = 3y$, which cannot be verified at compile-time. In such cases, before each invocation of a graph, code has to be generated, which will verify the balance equations for the current values of the parameters.

However, our pairwise clustering method for quasi-static scheduling naturally accommodates an efficient means for detecting sample rate inconsistencies. Whenever our clustering strategy encounters a pair of adjacent vertices *with more than one edge directed from one actor to the other*, consistency in the graph is equivalent to determining whether the ratio of the number of tokens produced to the number of tokens consumed is identical for all edges in that subgraph. Sample-rate consistency checks then leads to the setting up of a number of equations, which may be in terms of some unknown variables. In general, some of these equations can be verified symbolically at compile-time, and the rest must be verified at run-time, as a part of run-time local synchrony verification.

As an example, consider the PSDF graph G shown in Fig. 26. Here Ω_1 and Ω_2 represent two vertices in the graph, and may, in general, represent complex sub-

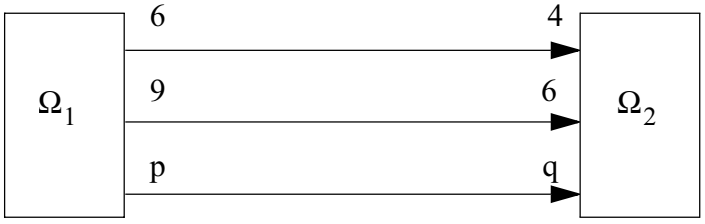


Figure 26. A PSDF graph that cannot be fully checked for consistency at compile-time.

graphs that have been consolidated through earlier clustering steps. The conditions for consistency of this subgraph are:

$$(6/4) = (9/6) = (p/q) \tag{66}$$

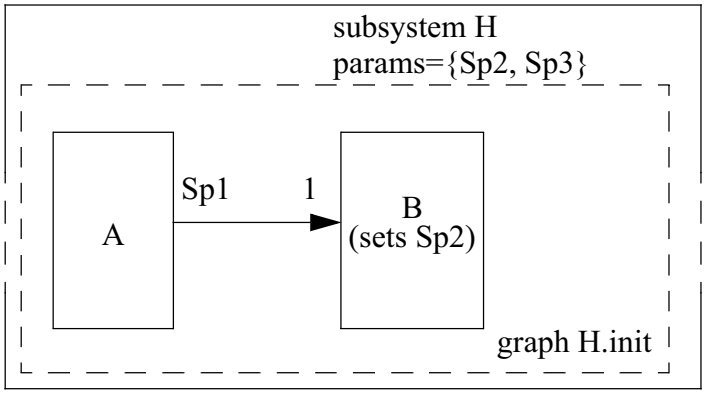
which gives us the single constraint for consistency:

$$(p/q) = (3/2) \tag{67}$$

This equation then has to be verified at run-time for consistency. If the parameters p, and q are such that (67) is always satisfied, then graph G is inherently dataflow consistent. If this equation does not hold for any permissible values of the parameters, then G is inherently dataflow inconsistent, otherwise G is partially dataflow consistent. For example, if $domain(p) = \{3, 15, 21\}$, and $domain(q) = \{2, 10, 14\}$, then G is inherently dataflow consistent; if $domain(p) = \{3, 15, 21\}$, and $domain(q) = \{5, 12, 10\}$; then G is inherently dataflow inconsistent, and if $domain(p) = \{3, 15, 21\}$, and $domain(q) = \{2, 12, 21\}$, then G is partially dataflow consistent.

It is straightforward to verify bounded memory consistency of a PSDF graph, which consists of checking that the max token transfer bounds are satisfied for the tokens consumed and produced onto each edge, and the max delay bounds are satisfied for every edge in the graph. If the token transfer or the delay on an edge is not statically known, then the quasi-static scheduler generates code that checks for these bounds at run-time.

For unit transfer consistency, each actor in the init (subinit) graph that is responsible for configuring the value of a parameter has to have a repetition count of one, and produce a single token at each of its output ports. If this cannot be verified at compile-time, then a conditional statement has to be inserted into the preamble code that appears before the init (subinit) graph schedule. An example is shown in Fig. 27. Part (a) of the figure shows the init graph of subsystem *H*, where actor *B* produces one token at its output port and configures the subsystem parameter *Sp2*. The repetition count of actor *B* depends on the subsystem inherited parameter *Sp1*, and the preamble code of the init graph contains a conditional for checking the value



(a)

```
/* preamble for H.init( ) */
if (Sp1 ≠ 1)
     error("unit transfer inconsistent")
/* body for H.init( ) */
fire A
repeat (Sp1) times {
     fire B
}
```

(b)

Figure 27. Quasi-static verification of unit transfer consistency.

of the repetition count at run-time, as shown in part (b) of the figure.

For local synchrony verification of the subinit input condition and the body condition, once the subinit and body graphs of a PSDF subsystem Φ are completely clustered by P-APGAN, the symbolic token flow computed on each hidden interface edge (see Section 4.3.1) of Φ_s and Φ_b is evaluated with default values of non-init-configured subsystem parameters of Φ, and these evaluated values are assigned as the token flow quantities on the corresponding edges in the parent graph H in which Φ is embedded. If for every interface edge, it is known at compile-time that the computed token flow does not depend on any non-init-configured parameter of Φ, then the subsystem Φ can be detected as locally synchronous at compile-time. Otherwise, code has to be inserted to compare the default token flow (as determined by evaluating the interface token flow with default values of non-init-configured parameters) with the actual token flow, where the actual token flow is evaluated with the current values of the non-init-configured parameters of Φ.

For example, consider a subsystem Φ with a body graph comprising two actors A and B as shown in Fig. 28. Suppose that the subsystem Φ is represented by the hierarchical actor H, the subinit graph of Φ does not have any interface ports, and the input edge to actor A is the only interface edge of Φ_b. After completely clustering Φ_b, the number of tokens consumed on the interface edge is computed as $(p_2)/(gcd(p_1, p_2))$. Now, if p_1 and p_2 are init-configured parameters of Φ, or they are compiler-generated variables representing unknown token flow, but the corresponding actors (actor A and actor B) have all their actor parameters

assigned either static values or init-configured subsystem parameter values, then it is clear at compile-time that the interface token flow of subsystem Φ does not depend on non-init-configured subsystem parameters and hence Φ can be detected as locally synchronous at compile-time. Otherwise code to check synchrony must be generated for Φ. If p_1 is an init-configured parameter, but p_2 is an internal parameter (subinit-configured) of Φ with a default value of 4, then the default token flow is equal to $(p_2)/(gcd(p_1, 4))$, and this will be assigned as the number of tokens consumed on the input interface edge of actor H. In addition, the synchrony-checking code will attempt to verify the equality

$$(p_2)/(gcd(p_1, p_2)) = (p_2)/(gcd(p_1, 4)), \tag{68}$$

which is equivalent to

$$(gcd(p_1, p_2))/(gcd(p_1, 4)) = 1. \tag{69}$$

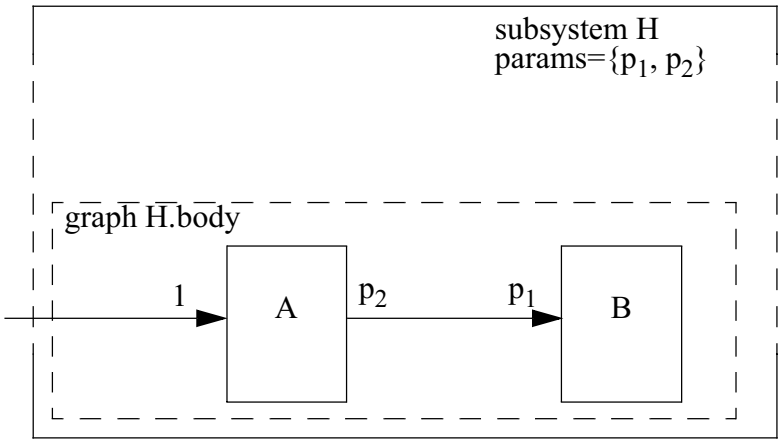


Figure 28. An example to demonstrate local synchrony verification issues in quasi-static scheduling.

Thus, whatever value p_2 takes on at run-time, if it maintains the same gcd relationship with p_1 as between p_1 and 4, then Φ will be locally synchronous, otherwise it will be flagged off as locally non-synchronous at run-time. It may be an intrinsic property of the application (unknown to the compiler) that the gcd of p_2 and p_1 is always equal to the gcd of p_1 and 4, in which case the system is inherently locally synchronous, otherwise it is partially locally synchronous or inherently locally non-synchronous.

4.3.3 User Assertions and Compile-time Predictability

As can be seen from the P-APGAN quasi-static scheduling technique, significant symbolic analysis and computation must, in general, be performed by the compiler. In the analysis process, at any point that the compiler does not have sufficient information to reach a conclusion, it generates code to perform the analysis at run-time. Similarly, in trying to compute a minimal periodic schedule of a PSDF graph, if the compiler lacks necessary data in the form of *gcd* information of symbolic token production and consumption quantities, then it uses compiler-assigned symbolic variables, and generates code to assign appropriate *gcd* values to those variables at run-time.

However, from a performance standpoint, the "leaner" the quasi-static schedule generated by the compiler, the better it is, both in terms of code size and run-time overhead. Thus, providing more information statically, allows more computation and analysis to be done at compile time, resulting in better performance. To provide more computation power to the compiler, the programmer can convey

application-specific knowledge to the compiler via *user assertions*. The following are examples of user assertion formats that a PSDF programmer can use.

Specify token flow at a PSDF actor port as a symbolic expression of the actor parameters (or as a static integer if it behaves as an SDF port), instead of providing a parameter interpretation function for the actor that computes the token flow at run time. Similarly, statically or symbolically specify the characteristics of a PSDF edge (amount of delay, initialization values, and re-initialization period), instead of providing a parameter interpretation function for that edge.

To enable *gcd* computation of the tokens produced and consumed onto an edge, as a part of the basic P-APGAN clustering step, specify *gcd* information among subsystem parameters, or specify product relationships among subsystem parameters by expressing a parameter as a product of other parameters, constants, and possibly some symbolic variables that do not represent parameters. For example, $p_1 = 2 \times a \times b \times p_2$, where p_1 and p_2 are subsystem parameters, while a and b are other symbolic variables that do not represent parameters. These product relationships can be used by the compiler to derive *gcd* information.

The flexibility of the PSDF model can be improved considerably if actor parameters can be identified that do not affect any of the port production or consumption quantities. A parameter $p \in params(A)$ is a *non-dataflow* parameter of A if for every input port $\rho \in in(A)$, $\kappa_A(\rho, x)$ is invariant over $\{p\}$, and for every output port $\rho \in out(A)$, $\varphi_A(\rho, x)$ is also invariant over $\{p\}$.

In general, it may not be always be possible to efficiently deduce or compute

the exact set of non-dataflow parameters of an actor. However, an actor designer usually knows whether or not a parameter is a non-dataflow parameter. The user can indicate the *known non-dataflow set (KNDS)* of A, denoted $KNDS(A)$, as a set of actor parameters that are known not to affect the dataflow behavior of the actor. For example, in the downsampler actor, the number of tokens consumed by the input port *in* is equal to the downsampling factor, while the output port *out* always produces one token. Thus, $KNDS(\text{dnSmpl}) = \{\text{phase}\}$.

Even more information can be provided to the compiler by precisely specifying which actor parameters control dataflow at each port of the actor. In general, the number of tokens consumed (or produced) at an input (or output) port θ of an actor A will depend on a subset of the actor parameters, called the *controlling set of* θ, denoted by $ctrl_A(\theta) \subseteq params(A)$. For example, in the downsampler actor, $ctrl_{\text{dnSmpl}}(\text{in}) = \{\text{factor}\}$, and $ctrl_{\text{dnSmpl}}(\text{out}) = \varnothing$. With the knowledge of control sets of an actor port, the compiler can do more sophisticated local synchrony analysis. For example, in Fig. 28, if p_1 is an init-configured parameter, p_2 is a compiler-generated variable representing unknown token flow, and actor A has two actor parameters — a_1 which is assigned an init-configured subsystem parameter, and a_2 which is assigned an internal (subinit-configured) subsystem parameter, then without any user assertions, the compiler cannot reach any conclusions about local synchrony of Φ at compile time. If however, the user asserts that the controlling set of the output port of actor A is given by $\{a_1\}$ then the compiler can determine at compile time that the subsystem is locally synchronous.

Such forms of user assertions deliver more power to the compiler and result in more compact schedules by decreasing the necessity of compiler-generated variables and compiler-generated conditional statements to perform consistency checks.

4.3.4 The Quasi-static Scheduling Algorithms

We will now present the complete scheduling algorithm, *compute_QS_schedule*, for quasi-static scheduling of PSDF graphs. The algorithm is shown in Fig. 29. Given a PSDF graph G, the scheduling is performed in a bottom-up fashion on a subsystem-by-subsystem basis. For every hierarchical actor in G, the init, subinit, and body graphs of the subsystem associated with that hierarchical actor are scheduled first, by recursively calling the *compute_QS_schedule* routine. After scheduling these three graphs, the unit transfer consistency constraints (if any) of the init graph and subinit graph are determined. The interface token flow of the subinit graph and body graph is evaluated with default values of non-init-config-

```
function compute_QS_schedule(graph G)
    foreach hierarchical actor H in G
        compute_QS_schedule(H.init( ))
        compute_QS_schedule(H.subinit( ))
        compute_QS_schedule(H.body( ))
        compute_unit_transfer_constr(H, H.init( ), H.subinit( ))
        compute_interface_token_flow(H, H.subinit( ), H.body( ))
    end for
    compute_bounded_memory_constr(G)
    initialize_for_clustering(G)
    compute_P_APGAN_schedule(G)
end function
```

Figure 29. The algorithm used for computing a quasi-static schedule for a PSDF graph.

ured parameters, and local synchrony constraints are generated, as necessary. The implementation of the quasi-static scheduler is streamlined by inserting the synchrony check code (VLS_s, and VLS_b) as a single block after the body graph execution, instead of inserting VLS_s before executing the subinit graph, and inserting VLS_b before executing the body graph.

Once each hierarchical actor has been processed in this fashion, the graph G is checked for bounded memory consistency, generating bounded memory constraints, as necessary. G is now ready to be clustered by P-APGAN. Prior to clustering, G is initialized by wrapping each actor in a cluster wrapper. Such initial clusters are referred to as "leaf" clusters, as opposed to "non-leaf" clusters that are produced after clustering steps. G is then clustered by the P-APGAN scheduling technique described in Section 4.3.1. In the course of P-APGAN clustering, sample rate consistency checks are performed, and requisite run-time constraints are generated. The scheduling process is initiated by calling the *compute_QS_schedule* routine with the top-level graph containing a single hierarchical actor representing the topmost subsystem.

The algorithm *traverse_QS_schedule* for traversing the quasi-static schedule generated by the *compute_QS_schedule* routine is shown in Fig. 30. The schedule is traversed in a top-down fashion, in accordance with the PSDF operational semantics (Section 3.7). The routine *traverse_QS_schedule* is called on a scheduled graph G with a list of (possibly empty) unit transfer consistency constraints for that graph. The algorithm involves traversing the cluster hierarchy generated by P-APGAN

scheduling. In the first traversal, the schedule of the init graph of each hierarchical actor in G is traversed by recursively calling the *traverse_QS_schedule* routine for every leaf cluster in G that contains a hierarchical actor. This is followed by processing in turn, all compiler-generated variables, all bounded memory consistency constraints, all unit transfer consistency constraints and all sample rate consistency

```
function traverse_QS_schedule(graph G, consistency_constr unit_trnsfr)
    foreach cluster C in cluster hierarchy of G
        if (is_leaf_cluster(C))
            if (is_hierarchical_actor(H = C.actor( )))
                traverse_QS_schedule(H.init( ), H.init_unit_trnsfr( ))
            end if
        end if
    end for
    process_compiler_generated_variables(G)
    process_bounded_memory_constr(G)
    process_unit_transfer_constr(unit_trnsfr)
    process_sample_rate_consistency_constr(G)
    foreach cluster C in cluster hierarchy of G
        if (is_leaf_cluster(C))
            if (is_hierarchical_actor(H = C.actor( )))
                traverse_QS_schedule(H.subinit( ), H.subinit_unit_trnsfr( ))
                traverse_QS_schedule(H.body( ), ∅)
                process_local_synchrony_constr(H.local_synchrony( ))
            else
                process_actor_firing(H)
            end if
        else
            process_cluster_reps(C)
        end if
    end for
end function
```

Figure 30. The algorithm for traversing the quasi-static schedule for a PSDF graph, generated by the *compute_QS_schedule* routine.

constraints for G. The next step involves once again traversing the cluster hierarchy of G. For a non-leaf cluster, the repetitions (local repetition factor) of that cluster is processed; for a leaf cluster containing a leaf actor, the actor firing is processed; and for a leaf cluster containing a hierarchical actor, the subinit graph schedule is traversed, followed by traversing the body graph schedule and finally processing the local synchrony constraints (if any), comprising the subinit input condition and the body condition for local synchrony of the subsystem represented by the hierarchical actor. As before, the quasi-static schedule traversal is initiated by calling *traverse_QS_schedule* with the top-level graph containing a single hierarchical actor representing the topmost subsystem.

We have presented a generic algorithm for traversing the quasi-static schedule generated by the *compute_QS_schedule* routine, which can be used as a framework for performing a variety of tasks — for example, printing the schedule, gathering statistics about the schedule, and synthesizing software based on the schedule. The different *process* routines present in the algorithm have to be specified accordingly. At present, we use this traversal algorithm for printing a quasi-static schedule, and the different *process* routines are specified to print the arguments passed to them in an appropriate format. Applications to synthesis would necessitate additional buffer memory management for the buffers on each edge of a PSDF graph, and a parameter configuration mechanism to associate each parameter with the value of the token generated at the actor output port where the parameter is configured. Arriving at an exact scheme to perform these additional synthesis tasks

appears to be a promising direction for further work

4.3.5 Re-initialization of Delays

Since PSDF extends SDF by allowing the same SDF system to be evaluated differently in different invocations by suitable assignment of the parameters, re-initialization of delays on graph edges becomes a necessary functionality. In the simplest case, conceptually, a PSDF system encompasses many runs of the underlying SDF system. Consider the Fibonacci Number example (Fig. 9), where by changing the value of p on the fly, the dataflow part of the system assumes different SDF configurations across multiple runs. Thus, it becomes necessary to re-initialize the values of the delay units on the feedback edges to the *add* actor, at the beginning of every run of the PSDF system.

One possible technique of inserting re-initializations at the proper point in the quasi-static schedule is the following. After scheduling the complete PSDF specification, examine each actor A that has an input edge specified with a re-initializable delay of re-initialization period p. Ascend the cluster hierarchy of the schedule, starting from actor A, until the *residual period* r either equals or becomes less than the local repetition factor at an ancestor cluster C. The residual period is initialized to p and is updated at every cluster R encountered on the path, by dividing the local repetition factor at that cluster ($reps(R)$) into the residual period, to obtain the new residual period. At the point of termination, if $reps(C)$ equals r, then insert the initialization such that while traversing the schedule, the initialization is processed before processing $reps(C)$. If $reps(C)$ exceeds r, then break it up into factors (if

possible), such that r is a factor. When traversing the schedule, process each factor of $reps(C)$ separately, with r coming last, and insert the initialization such that it is processed before processing the factor r of $reps(C)$.

However, in generating a looped schedule, if the re-initialization period does not coincide with a loop boundary, then it will not be possible to insert the initialization at the proper point in the schedule. In such cases, one possibility for the scheduler is to insert code at the beginning of the corresponding actor that keeps a count of the number of invocations of that actor, and initializes itself in a modulo fashion. But, our application design experience suggests that the re-initialization period often coincides with loop boundaries, as in the Fibonacci Number example (Fig. 9), and hence it is frequently possible to incorporate delay re-initializations into the quasi-static scheduling framework in a natural fashion.

4.3.6 Examples of Quasi-static Schedules

We will now present quasi-static schedules generated by our quasi-static scheduling algorithms (Section 4.3.4) for the PSDF examples of Section 2.4. The quasi-static schedules presented in this report do not include code for verifying the upper bounds specified by the max token transfer function at an actor port and the max delay value for an edge in a PSDF graph. This bound-checking code is straightforward, and has been omitted so as not to unnecessarily clutter the quasi-static schedules and instead, emphasize the more intricate issues.

The schedule, generated by our quasi-static scheduler, for the computation of the p th Fibonacci Number is shown in Fig. 31. The application has been fired for

five runs, and in each run it is possible to compute a different Fibonacci Number by suitably configuring the parameter p via the *setFib* actor. The delay values on the two input feedback edges of the *add* actor are re-initialized after every p invocations of *add*, for correct computation of the next Fibonacci Number. After p invocations of *add*, the Fibonacci Numbers from $F(1)$ to $F(p)$ are all lined up on the output of *add*. The single invocation of *dnSmpl* transmits the p th Fibonacci Number to its output, discarding the rest. The *print* actor then prints $F(p)$.

Fig. 32 shows the schedule derived by our quasi-static scheduling algorithms for computing the weighted average of variable length data packets. The application is fired for a hundred consecutive runs, processing hundred data frames generated by the *genData* actor. In each run, the *readHdr* actor configures the values of the packet length (parameter *pLen*) and the number of packets in a frame (parameter

```
repeat 5 times {
    /* begin init graph schedule for fib */
    fire setFib /* sets p */
    /* end init graph schedule for fib */

    /* begin body graph schedule for fib */
    initialize add
    repeat (p) times {
        fire add
    }
    fire dnSmpl
    fire print
    /* end body graph schedule for fib */
}
```

Figure 31. The quasi-static schedule for the Fibonacci Number computation example of Fig. 9.

fLen). In *pLen* invocations, the *mult* actor multiplies together all the data values in a single packet, accumulating the intermediate products at its output. One invocation of the *dnSmpl1* actor transmits the final product to the input of the *add* actor. The input of the *mult* actor is then re-initialized to compute the running product of the next data packet. In *fLen* invocations, the *add* actor adds up the computed products of all the data packets present in a single frame. The *dnSmpl2* actor provides the final sum to the *div* actor which divides it by the frame length to obtain the weighted average of a single data frame. The input of the *add* actor is then re-initialized to

```
repeat 100 times {
    /* begin init graph schedule for wtAvg */
    fire genHdr
    fire readHdr /* sets fLen, pLen */
    /* end init graph schedule for wtAvg */

    /* begin body graph schedule for wtAvg */
    initialize add
    repeat (fLen) times {
        initialize mult
        repeat (pLen) times {
            fire genData
            fire mult
        }
        fire dnSmpl1
        fire add
    }
    fire dnSmpl2
    fire div
    fire print
    /* end body graph schedule for wtAvg */
}
```

Figure 32. The quasi-static schedule for computation of the weighted average of variable length data packets, specified in Fig. 11.

process the next data frame.

Fig. 33 shows the schedule for the predictor application, for five hundred runs. In a single run, the step size (parameter *step*) and filter length (parameter *fLen*) of the adaptive filtering mechanism are set to a certain value, and a thousand samples of a random signal are processed through the adaptive filter, which tries to predict the future value of the signal based on the present value. The *wtCntrl* actor adapts the filter coefficients (parameter *coeffs*) of the *FIR* actor after processing

```
repeat 500 times {
    /* begin body graph schedule for predictor */

    /* begin init graph schedule for adaptFilt */
    fire setPars /* sets step, fLen */
    /* end init graph schedule for adaptFilt */

    fire randSig
    repeat (1000) times {
        fire fork1
        fire fork2

        /* begin subinit graph schedule for adaptFilt */
        fire wtCntrl /* sets coeffs */
        /* end subinit graph schedule for adaptFilt */

        /* begin body graph schedule for adaptFilt */
        fire FIR
        fire Subtract
        /* end body graph schedule for adaptFilt */

        fire fork3
        fire plot
    }
    /* end body graph schedule for predictor */
}
```

Figure 33. The quasi-static schedule for the predictor application of Fig. 12.

every sample. It is possible to fine tune the step size and filter length across multiple runs of the application to obtain the best possible prediction result.

4.4. Run-time Scheduling

Run-time scheduling is performed for those PSDF specifications for which quasi-static schedules cannot be computed, or only partial quasi-static schedules can be computed (i.e. portions of the graph can be clustered). The run-time scheduling algorithm *compute_RT_schedule* is presented in Fig. 34, and is a straightforward implementation of the PSDF operational semantics. The function is called on a PSDF graph G with a table of parameter values *param_table*. The entries in *param_table* represent the current snapshot of known parameter values at this point of execution in the program. Since partial quasi-static scheduling may have been done at compile-time through P-APGAN, G may consist of normal actor vertices, and cluster vertices. The latter appear just as actor vertices, except that they contain cluster hierarchies inside them. The first step comprises visiting every hierarchical actor H in G (present as an actor vertex, or inside the cluster hierarchy of a cluster vertex), and scheduling the init graph of H by recursively calling *compute_RT_schedule*.

If the hierarchical actor H is present inside a cluster hierarchy, then the default interface token flow of H has already been computed by P-APGAN. Otherwise, it is computed now in the routine *compute_interface_tokens* as shown in Fig. 35. Given the hierarchical actor H, this function first computes the interface token flow of every hierarchical actor present in the subinit and body graphs of H by

```
function compute_RT_schedule(graph G, param_values param_table)
    foreach node A in G
        if (is_cluster(A))
            foreach cluster C in cluster hierarchy of A
                if (is_leaf_cluster(C))
                    if (is_hierarchical_actor(H = C.actor( )))
                        compute_RT_schedule(H.init( ), param_table)
                    end if
                end if
            end for
        else if (is_hierarchical_actor(A))
            compute_RT_schedule(A.init( ), param_table)
            compute_interface_tokens(A, param_table)
        end if
    end for
    configure_graph_as_SDF(G, param_table)
    compute_SDF_repetitions(G)
    S = construct_valid_schedule(G)
    while (L = get_next_firing(S))
        if (is_hierarchical_actor(L))
            configure_parent_configured_params(L, param_table)
            compute_RT_schedule(L.subinit( ), param_table)
            compute_RT_schedule(L.body( ), param_table)
            verify_interface_token_flow(L, param_table)
            delete_params(L.internal_params( ), param_table)
            delete_params(L.parent_configured_params( ), param_table)
        else if (is_cluster(L))
            cluster_sched = invocation_sequence(L)
            replace_in_schedule(S, L, cluster_sched)
        else
            fire_actor(L)
            foreach parameter p set by L to value v
                insert_param(p, v, param_table)
        end if
    end while
    foreach hierarchical actor H in G
        delete_params(H.init_configured_params( ), param_table)
    end for
end function
```

Figure 34. The algorithm for computing a run-time schedule for a PSDF graph.

recursively calling itself. Once all the hierarchical actors are taken care of, it configures the subinit and body graphs of H as SDF graphs in the routine *configure_graph_as_SDF*. This is done in two steps. In the first step, integer values are assigned to $p(e)$, $c(e)$ and $d(e)$ of each edge e in the graph (including interface edges) by evaluating symbolic expressions or unknown token flow (via parameter interpretation functions), using current parameter values. If a parameter occurs in *param_table* — this will be true for all init-configured and inherited parameters of H — then the associated value is obtained from *param_table*, otherwise the default value of the parameter is used. In the second step, the cluster hierarchy of every cluster vertex present in the graph is traversed and all the symbolic expressions in each cluster C are evaluated to an integer value in an analogous manner. These include the local repetition factor ($reps(C)$) of a cluster, the associated sample rate consistency constraints, the associated bounded memory consistency con-

```
function compute_interface_tokens(subsystem H, param_values param_table)
    foreach hierarchical actor A in H.subinit( )
        compute_interface_tokens(A, param_table)
    end for
    foreach hierarchical actor A in H.body( )
        compute_interface_tokens(A, param_table)
    end for
    configure_graph_as_SDF(H.subinit( ), param_table)
    compute_sdf_repetitions(H.subinit( ))
    configure_graph_as_SDF(H.body( ), param_table)
    compute_sdf_repetitions(H.body( ))
    adjust_and_store_interface_token_flow(H)
end function
```

Figure 35. The algorithm for pre-computing the interface token flow of a hierarchical actor in a PSDF graph, used in run-time scheduling of the PSDF graph.

straints, and for a leaf cluster containing an hierarchical actor, the default interface token flow. After this operation, the body graph and subinit graphs of H have assumed (local) SDF configurations. The next step is to compute the repetitions vector of each graph. This is done in the routine *compute_SDF_repetitions*, by a *depth-first search* algorithm as discussed in [8]. Once the repetition counts of the interface actors are known, the interface token flow of H is computed, and stored for future local synchrony verification.

At this point, the init graphs of all hierarchical actors in G have been fired, and their interface token flows have been pre-computed. It remains to determine a schedule for G and execute it. G is configured as an SDF graph by calling the routine *configure_graph_as_SDF* as explained above. Note that in this case, all parameters will have known values in *param_table*, and default values will not be needed, as now we are doing the actual scheduling, as opposed to speculative scheduling in the case of interface token flow pre-computation. The SDF repetitions vector of G is computed, and a valid schedule S is constructed by applying a class-S algorithm proposed by Lee [26] and given in [8]. S consists of a list of firings of graph vertices, and the next step is to execute these vertex invocations. Each actor L is extracted from S in order, and is processed as follows:

If L is a non-hierarchical actor in G, then we simply fire (execute) it. If the actor sets any parameters at its output ports, then those parameter entries are inserted into the *param_table*.

If L is a hierarchical actor, then we first configure any parent-configured

parameter of L bound to an input edge of L by inserting its value into the *param_table*. This is followed by scheduling and executing the subinit graph of L, and then the body graph of L. After the subinit and body graphs have been scheduled and executed, we have actual values for the interface token flow of L. These are compared with the pre-computed values in the routine *verify_interface_token_flow*, and in case of any mismatch, a local non-synchrony error is flagged off, and execution is terminated. This completes one invocation of subsystem L, and its internal parameters and parent-configured parameters are removed from *param_table*. These parameters will be set to fresh values in the next invocation of L.

If L is a cluster vertex, then the cluster hierarchy present inside L is traversed to obtain an equivalent invocation sequence corresponding to the looped schedule represented by the cluster hierarchy. For example, the looped schedule $5((2A)B)$ is equivalent to the invocation sequence $AABAABAABAABAAB$. The firing of L is replaced in S with the equivalent invocation sequence, so that in the next iteration the first member of this sequence will be extracted.

After the schedule S is exhausted, one invocation of G is completed, and the init-configured parameters of each hierarchical actor in G are removed from *param_table*. These parameters will be assigned new values in the next invocation of G.

In addition to local synchrony verification (verifying the subinit input condition and the body condition for local synchrony of a child subsystem of G), the run-

time scheduler must also check for deadlock, sample rate consistency, and bounded memory consistency of G. Also, if G is an init graph or subinit graph of its parent subsystem, then the run-time scheduler must verify the init condition or the subinit output condition for local synchrony of the parent subsystem of G. The bounds specified by the max token transfer function and the max delay value are verified after the graph has been configured as a (local) SDF graph in *configure_graph_as_SDF*. Sample rate consistency and unit transfer consistency (init condition or subinit output condition) are checked in *compute_SDF_repetitions*. For non-clustered portions of the graph, these are detected while constructing the repetitions vector of the graph. For clustered portions of the graph, these checks are conducted by traversing each cluster hierarchy after computing the repetitions vector. Sample rate constraints are already associated with a cluster, and these just needs to be verified. For unit transfer consistency, the number of tokens produced at an output port of each actor contained in a leaf cluster is computed and compared with one if the actor configures any parameter of the parent subsystem at that output port. Deadlock detection occurs while constructing a valid schedule in *construct_valid_schedule*. Given the repetitions vector $\mathbf{q}$, our class-S scheduling algorithm (*construct_valid_schedule*) maintains the state of the system, and repeatedly schedules fireable actors, updating the system state as each actor is fired, until all actors have been scheduled exactly the number of times specified by the corresponding component of $\mathbf{q}$, or until no actor is fireable. In case of the latter, deadlock is detected. Recall that quasi-static scheduling (through cluster-

ing) is done only in acyclic portions of a graph, so deadlock analysis need not be done inside clusters present in the graph.

Re-initializable delays present on an edge e can be handled by keeping a count of the number of executions of $snk(e)$ in the "while" loop of Fig. 34, and initializing the delay on e after every p executions of $snk(e)$, where p is the re-initialization period.

Fig. 36 shows the trace generated by the run-time scheduler for the Fibonacci Number example (Fig. 9). The application is run twice, computing the 5th and 7th Fibonacci Numbers respectively.

4.5. Implementation

We have implemented a PSDF tool that accepts PSDF specifications written in a specific textual format, performs quasi-static scheduling and prints a quasi-static schedule. When a quasi-static schedule cannot be constructed, our tool performs run-time scheduling, producing an output schedule trace of actor firings. All the schedules presented in this report are automatically generated from this tool, except for a few features explained below. Work is under progress to incorporate various enhancements into this tool.

4.5.1 Current Status

In this section, we document the current status of the implementation. Re-initialization of delays is not yet a part of the tool. At present quasi-static scheduling is performed only for acyclic graphs, where all the token flow is specified either in

terms of symbolic expressions of subsystem parameters, or as static integers, and there is no unspecified token flow to be obtained at run-time from a parameter interpretation function. We refer to the latter as "statically unspecified token flow". Cyclic graphs and graphs with statically unspecified token flow are passed on for run-time scheduling. For specifications with a significant amount of statically

```
Firing 1 {
    /* scheduling init graph of fib */
    fire setFib /* p  set to value 5 */

    /* scheduling body graph of fib */
    init add
    fire add
    fire add
    fire add
    fire add
    fire add
    fire dnSmpl
    fire print
}

Firing 2 {
    /* scheduling init graph of fib */
    fire setFib /* p  set to value 7 */

    /* scheduling body graph of fib */
    init add
    fire add
    fire add
    fire add
    fire add
    fire add
    fire add
    fire add
    fire dnSmpl
    fire print
}
```

Figure 36. The schedule trace generated by the PSDF run-time scheduler for the Fibonacci Number example of Fig. 9.

unspecified token flow, it appears that the quasi-static scheduler is left with too little static information, so that the resulting schedule has a lot of baggage in it, leading to inefficiency.

The pre-processor described in Section 4.2 is under development. Currently, for simple cyclic graphs, cycles are manually broken in the input specification (i.e. the feedback edge is not specified) before being presented to the tool. Taking actor markings into consideration, while clustering in P-APGAN (as described in Section 4.3.1) has not yet been implemented.

Our tool currently implements either a quasi-static scheduling strategy, based on a parameterized looped schedule, or a fully dynamic scheduling strategy, without any attempt to construct hybrid schedules involving both approaches. Thus, there is no provision for the run-time scheduler to handle portions of the specification for which quasi-static schedules have already been developed at compile-time. The trade-offs between the two scheduling strategies in the PSDF context, and their relative degree of usage in a combined model are interesting areas for further research. However, the advantages of clustering portions of the graph about which enough static information is present (e.g. SDF subgraphs, single rate subgraphs, subgraphs for which symbolic *gcd* information is known through user assertions), as done in the BDF model [10], are obvious. So, we are working on moving from the "either-or" scenario to a "as-much-static-clustering-as-possible" scenario, and having the run-time scheduler efficiently handle these clustered vertices, as outlined in the run-time scheduling algorithm *compute_RT_schedule* (Section 4.4).

At present, we do not have a library of actors that provide the code for executing the functionality of an actor. This does not affect the quasi-static scheduler, as the scheduling process is independent of the internal execution of an actor. However, the run-time scheduler does depend on those actor executions that are responsible for configuring the value of a subsystem parameter. In the absence of an actor library, our tool reads parameter values from an input file (with a *.param* extension), and performs a simulation of the application based on the schedule constructed with the parameter values read from the file.

4.5.2 C Implementation using LEDA

The PSDF tool has been implemented in the C programming language on the Solaris platform. The implementation uses the LEDA [27] software, which provides a library of data types and algorithms of combinatorial computing. LEDA uses C++ to implement the various data types and combinatorial algorithms that it provides. Hence we have used some C++ constructs (e.g., the *new* operator) in conjunction with LEDA data types, and the *g++* compiler has been used for compilation purposes. We have included selected sections of the code for the PSDF tool in the Appendix of this thesis.

We use the *Parameterized Graphs* (GRAPH) data type available in LEDA. A parameterized graph G is a graph (directed multigraph in our case) whose nodes and edges contain additional (user defined) data. Every node contains an element of a data type *vtype*, called the node type of G, and every edge contains an element of a data type *etype*, called the edge type of G. We have used the pre-defined LEDA

subroutines for creating a parameterized graph, accessing and traversing the graph (e.g., obtain the first node of a graph, iterate over all edges of the graph), updating the graph (e.g., adding new vertices and edges, hiding or removing an edge), and for ascertaining various properties of the graph (e.g., if it is cyclic or acyclic). In our implementation, we have employed two different types of parameterized graphs, *actor graph*, and *cluster graph*, differentiated by their node types (vtype). In an actor graph G, the node type of G (vtype), called *attr_actor* is a structure containing a single field "alist" (*attribute list)*, which is defined as a pointer to a hash table. The edge type of G (etype) is called *attr_arc* and is similarly defined. In a cluster graph G, the node type of G is called *attr_cluster*, and is also defined analogously. The attribute list (i.e. the hash table) consists of a collection (list) of *attributes*, where each attribute is a tuple of {*attribute-name*, *attribute-value*}. Each attribute-name is an enumerated data type representing a particular property of the object with which it is associated (actor, edge or cluster). For example, relevant attributes of a PSDF actor include the actor name, its parameter list, the number of actor ports and its hierarchy property (whether it is a leaf actor, or a hierarchical actor), while for a PSDF edge, the edge delay appears as an attribute. For a PSDF cluster, some of the attributes are the number of child members in the cluster (one for a leaf cluster, and two for a non-leaf cluster), the local repetition factor of the cluster, and the *gcd* variable (if any) generated by the compiler while creating the cluster from a P-APGAN clustering step. As mentioned above, the collection of all the attributes of an object is maintained in a hash table, where the attribute-name serves as the key,

and the attribute-value serves as the associated data. The hash table has been implemented in a generic fashion such that any data type can be stored as the attribute-value associated with an attribute-name, after being typecast as a *void** pointer. The interface to every attribute has been implemented uniformly as a set of five routines — *create*, *set*, *get*, *delete*, and *print*, where the attribute-name appears as the prefix of each routine. For example, for the actor attribute-name *actor_attr_numports*, these five routines will appear as *actor_attr_numports_create, actor_attr_numports_set*, *actor_attr_numports_get*, *actor_attr_numports_delete*, and *actor_attr_numports_print*. Handling the hash table appropriately for each attribute, is abstracted into these five routines, and the higher-level routines can simply utilize this interface. In fact, if the attribute-name, the data type of the attribute-value, and some other properties of the attribute are provides as parameters, then it is straightforward to automatically generate these set of five routines. The Appendix includes the file "attr.h", which defines the data types attr_actor, attr_arc, and attr_cluster. The file "actor_attr.h" shows an enumeration of some of the attributes of a PSDF actor, while the two files "actor_attr_numports.h" and "actor_attr_numports.c" provide a specification of the five interface routines for the *actor_attr_numports* attribute-name of an actor attribute.

The quasi-static scheduling algorithms in the PSDF tool perform significant symbolic computation. This symbolic computation has been abstracted into two C files, "psdf_polynomial.h" and "psdf_polynomial.c", that implements an abstract data type *psdf_polynomial*. A psdf_polynomial comprises a symbolic expression

appearing, in general, as a fraction, where the numerator and denominator is represented as a product of parameters (symbolic variables), and integers. The header file (psdf_polynomial.h) declares the interface used to access a psdf_polynomial, in the form of subroutines (functions) that include creating a psdf_polynomial from a parameter, performing various computations on two psdf_polynomials (e.g., multiplication, division, *gcd* computation), ascertaining various properties about a psdf_polynomial (e.g., if it is an integer), and printing a psdf_polynomial in an appropriate format.

The input to the PSDF tool is a PSDF specifications written in a specific textual format. Fig. 37 shows an example of a more detailed model (including explicit fork actors) of the Fibonacci Number computation application (Fig. 9) specified in this textual format. The "#" sign beginning a line denotes a comment line. Each subsystem is specified with the keyword $SUBGRAPH followed by the name of the subsystem. Subsystem parameter information is provided next, with specifics about each parameter, including its name, classification (e.g., external-dataflow), data type, and default value. *gcd* characteristics and product relationships among parameters (see Section 4.3.3) can also be specified. The init, subinit, and body graphs are enumerated with the keywords $INIT, $SUBINIT, and $BODY, respectively. Within each graph, the number of actors, edges and child subsystems is provided. The format for instantiating the actors in a graph uses the $ACTOR-INSTANCES keyword. Each actor is specified by the "actor" keyword followed respectively, by the name of the actor instance, the name of the actor in the actor library, and optionally the num-

```
# A simple application demonstrating computation of the pth fibonacci number.
# the topmost sub-system
$SUBGRAPH fib
{
numparameters 1
external-dataflow parameter p
type int
default 5
$INIT
{
numactors 1
numedges 0
numsubgraphs 0
$ACTOR-INSTANCES
actor setPar userInput sets 1
sets p
}
$BODY
{
numactors 5
numedges 6
numsubgraphs 0
$ACTOR-INSTANCES
actor add Add
actor fork1 Fork
actor fork2 Fork
actor dnSmpl DnSmpl actor-params 2
actor-param factor type int assigned sub-param p
actor-param phase type int assigned static 0
actor print Print
$CONNECTIVITY
connect add fork1 1 1 0
connect fork1 fork2 1 1 0
connect fork1 add 1 1 1
connect fork2 add 1 1 2
connect fork2 dnSmpl 1 p 0
connect dnSmpl print 1 1 0
}
}
```

Figure 37. A more detailed representation of the Fibonacci Number example of Fig. 9, specified in the input textual format of the PSDF tool.

ber of actor parameters (using the keyword “actor-params”) and the number of subsystem parameters that are configured at output ports of the actor (using the keyword “sets”). Each actor parameter is specified on a separate line with its data type and its configuration (either with a static value or with a subsystem parameter). If an actor configures a parameter, then that is specified on a separate line with the name of the configured parameter. Edges in a graph are enumerated with the $CONNECTIVITY keyword. Each edge e is represented as connecting the source actor instance with the sink actor instance, followed by the number of tokens produced onto the edge ($p(e)$), the number of tokens consumed from the edge ($c(e)$)and the delay on the edge ($d(e)$). A negative value of “-1” for $p(e)$, $c(e)$, or $d(e)$ implies that the quantity is statically unspecified, instead, parameter interpretation functions have been provided. When a child subsystem is embedded in a graph, interface connectivity information is provided in the subsystem specification using the $INTERFACE keyword. Each interface edge is labelled with its direction (input or output) and connects an actor instance in the parent graph with an actor instance in the associated init, subinit, or body graph of the subsystem. Values for $p(e)$, $c(e)$, and $d(e)$ are provided for an interface edge, similar to a normal edge in a graph.

The executable of the PSDF tool is called *psdf_sched*, and the *main* routine is specified in the file “psdf_sched.c”, provided in the Appendix. The tool accepts as a command line argument, the name of a file with a *.psdf* extension, containing a PSDF specification in the textual format described above. The number of firings of the PSDF specification in the generated schedule can also be provided as an optional

command-line argument. The textual format present in the *.psdf* file is parsed in the "psdf_read_graph.c" file, and the specification is converted into an actor_graph. The code for the run-time scheduler is implemented in the file "psdf_run_time_schedule.c" and it operates on the actor_graph structure obtained after parsing the input file. For quasi-static scheduling, the specification is converted into a cluster_graph from an actor_graph, by providing a cluster wrapper around each actor vertex. The quasi-static scheduling algorithms operate on the cluster_graph, and are specified in the file "psdf_schedule.c". Initializing the cluster hierarchy, pairwise clustering of adjacent nodes in P-APGAN, and traversing the cluster hierarchy for printing a quasi-static schedule are implemented in the "psdf_cluster.c" file. The Appendix contains selected sections of code from the C files "psdf_schedule.c", "psdf_cluster.c", and "psdf_run_time_schedule.c". The routines defined in psdf_read_graph.c, psdf_schedule.c, psdf_cluster.c, and psdf_run_time_schedule.c use the psdf_polynomial interface specified in "psdf_polynomial.h", and the five attribute interface routines of the different PSDF actor, edge, and cluster attributes, as specified in the corresponding header files of the attributes.

The output of the PSDF tool is the file "output.sched" which contains a schedule (quasi-static or generated at run-time) in a pseudo-code format for the input PSDF specification. All the schedules shown in this thesis (e.g., Fig. 31, 36) follow the output format of the PSDF tool.

Chapter 5. DSP Applications in PSDF

In this chapter, we present three DSP applications, modeled as PSDF representations, and the quasi-static schedules obtained for each application by our quasi-static scheduling algorithms (Section 4.3). In representing the applications, we have omitted specifying the upper bounds represented by the max token transfer function at an actor port, and the max delay value on an edge. This has been done so as not to clutter the representation, and instead emphasize the salient features about PSDF modeling of the application functionality. From our experience, it appears that specifying an appropriate bound (for the token transfer and delay), based on the programmer's application-specific knowledge, is usually a straightforward task in the design process.

5.1. Speech Compression

In this section, we will describe a speech compression application modeled in the PSDF framework. A speech sample is to be transmitted from the sender side to the receiver side using as few bits as possible, applying *analysis-synthesis* [20, 21] techniques. On the sender side, the speech sample is first analyzed through linear predictors to obtain the corresponding auto-regressive (AR) coefficients of the sample [20, 21]. To obtain an AR model, we require the speech sample to be wide sense stationary (WSS) such that the spectral characteristics of the process remain constant over the duration of the sample length. Thus, it is necessary to break up the

speech sample into small segments spanning a duration of a few milliseconds or less, over which the signal may be assumed to be approximately stationary. A separate AR model is then used for each segment. Another important design issue is choosing the model order of the AR process. Once the optimal model order and segment size have been determined, the speech segment is analyzed through linear predictors to obtain the AR coefficients, and the residual error signal. These are then quantized and transmitted to the receiver side, where the residual and the coefficients are dequantized, followed by AR modeling to reconstruct the original speech segment. For a good AR model, the dynamic range of the residual error signal is much smaller than that of the original signal. So this approach allows us to use fewer bits to transmit the speech signal, compared to direct transmission.

5.1.1 PSDF Representation — *Style 1*

Fig. 38 shows one possible representation of the speech compression application modeled by the PSDF specification *Compress*. The length of a speech instance L is an external parameter of this subsystem, that is set in the init graph. 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. The *Speech1* and *Speech2* actors are responsible for generating samples of this speech instance (e.g, via interface to an A/D converter). The segment size N, the model order M, and the zero-padded speech sample length R are internal parameters of the *Compress* subsystem. In the subinit graph, the *Select* actor reads the entire speech instance, and

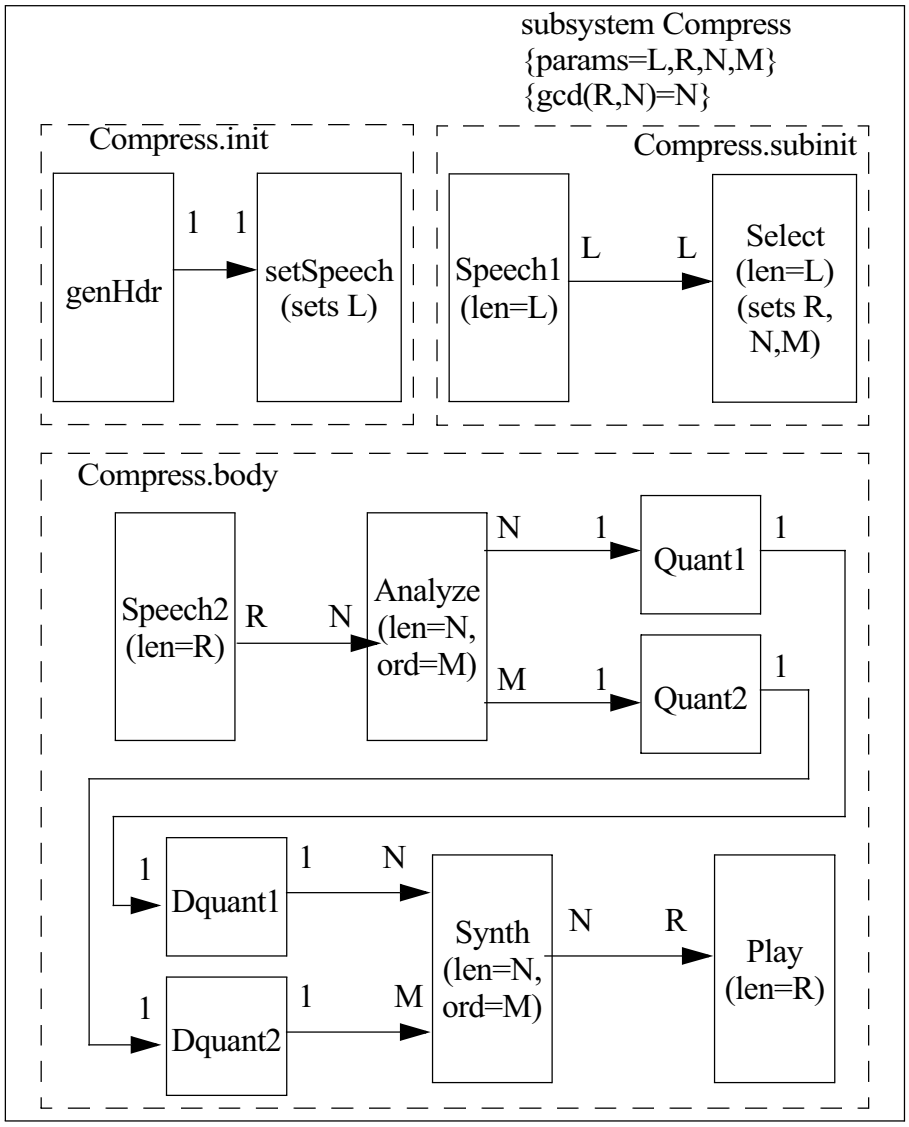


Figure 38. The Speech Compression application modeled in PSDF: *Style 1*.

examines it to determine the model order and segment size, using any of the existing techniques, e.g. the Burg segment size selection algorithm, and the AIC order selection criterion [21]. The zero-padded speech length R is determined such that it is the smallest integer greater than L satisfying $mod(R, N) = 0$, i.e. the segment size divides the zero-padded speech sample length exactly. This fact is conveyed to the scheduler through the user assertion $gcd(R, N) = N$.

In the body graph of the specification, the upper half of the diagram represents the transmitter side, and the lower half represents the receiver side. In the transmitter side, the *speech2* actor generates the speech sample, zero-padding it to a length R. The *Analyze* actor accepts speech segments of size N, and performs linear prediction on the speech segment, producing M AR coefficients and the residual error signal of length N at its output. The model order (*ord*) and input length (*len*) actor parameters of the *Analyze* actor are assigned the subsystem parameters M and N, respectively. Each sample of the residual signal is quantized and encoded by a scalar quantizer (*Quant1*) and transmitted to the receiver side where it is dequantized by the *Dquant1* actor. The AR coefficients are also transmitted and received in a similar fashion.The *Synth* actor reconstructs each speech segment by performing AR modeling, using the M AR coefficients and the residual signal of length N as excitation. Finally the *Play* actor first allows all the segments to accumulate at its input, then accepts R samples of the entire speech instance, and plays the resulting audio segment.

Note that for clarity, the above PSDF model does not specify all the details

of the application. Our purpose is to give an overview of the modeling process, and concentrate on those parameters that are relevant from the scheduler's perspective. The model does not go into the details of the *Select* actor, which will most likely be a subblock (representing syntactic hierarchy as opposed to semantic hierarchy) consisting of other actors inside it. Also, we have omitted all parameters that do not affect the dataflow behavior of the application. For example, the *Speech1* and *Speech2* actors are treated as "black boxes", without going into the specifics of how exactly the speech instance is generated. One possible scheme is reading from a file (like a *.au* file), in which case *speechFile* could be a parameter of the specification, specified as a part of the header packet, and configured in the *setSpeech* actor, similar to the parameter L. Such a file-based interface is used, for example in Ptolemy [12]. Under such an interface, *Speech1* and *Speech2* could be instances of an actor with two parameters — *fileName* (set to subsystem parameter *speechFile*) and *len*, where the functionality is to read a speech instance from the file, and produce *len* samples of it. If *len* is less than the length of the speech instance in the file, then it is truncated, otherwise it is zero-padded. Similarly, the quantizers and dequantizers will have actor parameters controlling their quantization levels, thresholds, etc. The *Select* block could determine two such sets — one for the residual and one for the coefficients, and set them up through internal subsystem parameters, which could then be assigned to the actor parameters. In an alternate specification, the init graph could consist of a single actor that configures the different characteristics of a speech instance (e.g., length, name of the associated file) from user input.

The quasi-static schedule generated for the *Compress* specification by our quasi-static scheduler is shown in Fig. 39. The application is run five times, and in each run a different speech instance can be processed. The schedules for the init and subinit graphs are straightforward, with single invocations of every actor. In the body graph, the *Analyze* and *Synth* actors process N samples of the speech instance

```
repeat 5 times {
    /* begin init graph schedule for Compress */
    fire genHdr
    fire setSpeech /* sets L */
    /* end init graph schedule for Compress */

    /* begin subinit graph schedule for Compress*/
    fire Speech1
    fire Select /* sets R, N, M */
    /* end subinit graph schedule for Compress*/

    /* begin body graph schedule for Compress*/
    fire Speech2
    repeat (R/N) times {
        fire Analyze
    }
    repeat (R) times {
        fire Quant1
        fire Dquant1
    }
    repeat (M×(R/N)) times {
        fire Quant2
        fire Dquant2
    }
    repeat (R/N) times {
        fire Synth
    }
    fire Play
    /* end body graph schedule for Compress*/
}
```

Figure 39. The quasi-static schedule for the Speech compression application (*Style 1*) of Fig. 38.

in each invocation. So, they are scheduled for R/N invocations to process all R samples of the speech instance. Each invocation of the *Analyze* actor generates N samples (each of which is processed by the *Quant1* and *Dquant1* actors) of the residual signal, and M AR coefficients (each of which is processed by the *Quant2* and *Dquant2* actors). Thus the total number of invocations of the *Quant1* and *Dquant1* actors is $(N \times (R/N))$ which is equal to R, and the total number of invocations of the *Quant2* and *Dquant2* actors is $(M \times (R/N))$. This schedule is generated when the compiler knows that N divides R exactly (via a user-specified assertion). Without this knowledge, the scheduler must introduce compiler-generated variable to represent the *gcd* of R and N. The management of this variable complicates the schedule as shown in Fig.40, which gives an alternative schedule in which it is not known that N divides R exactly.

5.1.2 PSDF Representation — *Style 2*

An alternate PSDF model (Style 2) of the speech compression application is presented in Fig. 41. In this model, the analysis and synthesis of the speech sample are abstracted into a separate subsystem *AnalyzeSynthesize* embedded in the topmost subsystem *Compress*. The init graph of *Compress* sets up the length L of a speech instance, which is generated once by the *Speech* actor in the body graph and passed on to the *AnalyzeSynthesize* subsystem for processing. Both the subinit and body graphs of *AnalyzeSynthesize* receive the speech sample as dataflow input. The segment size N, the model order M, and the zero-padded length R have now become parameters of the *AnalyzeSynthesize* subsystem. These parameters are con-

figured in the subinit graph by the *Select* actor as before. An additional parameter K has been introduced in the *AnalyzeSynthesize* subsystem. The need for this parameter, and the appropriate place to configure it are somewhat subtle. The *zeroPad* actor in the body graph receives the speech sample of length L from the parent graph, zero-pads it to length R and passes it on to the *Analyze* actor. However, unlike the subinit graph, the body graph cannot specify L as the number of tokens consumed at

```
/* begin body graph schedule for Compress*/
/* gcd variable declarations */
int _gcd_1 = gcd(R, N)
int _gcd_2 = gcd((N × R)/ _gcd_1, R)
repeat (R/ _gcd_2) times {
    repeat (N/ _gcd_1) times {
        fire Speech2
    }
    repeat (R/ _gcd_1) times {
        fire Analyze
        repeat (N) times {
            fire Quant1
            fire Dquant1
        }
        repeat (M) times {
            fire Quant2
            fire Dquant2
        }
        fire Synth
    }
}
repeat ((N × R)/(_gcd_1 × _gcd_2)) times {
    fire Play
}
/* end body graph schedule for Compress*/
```

Figure 40. The quasi-static schedule for the body graph of the *Compress* subsystem of Fig. 38, where the compiler is unaware that N divides R exactly.

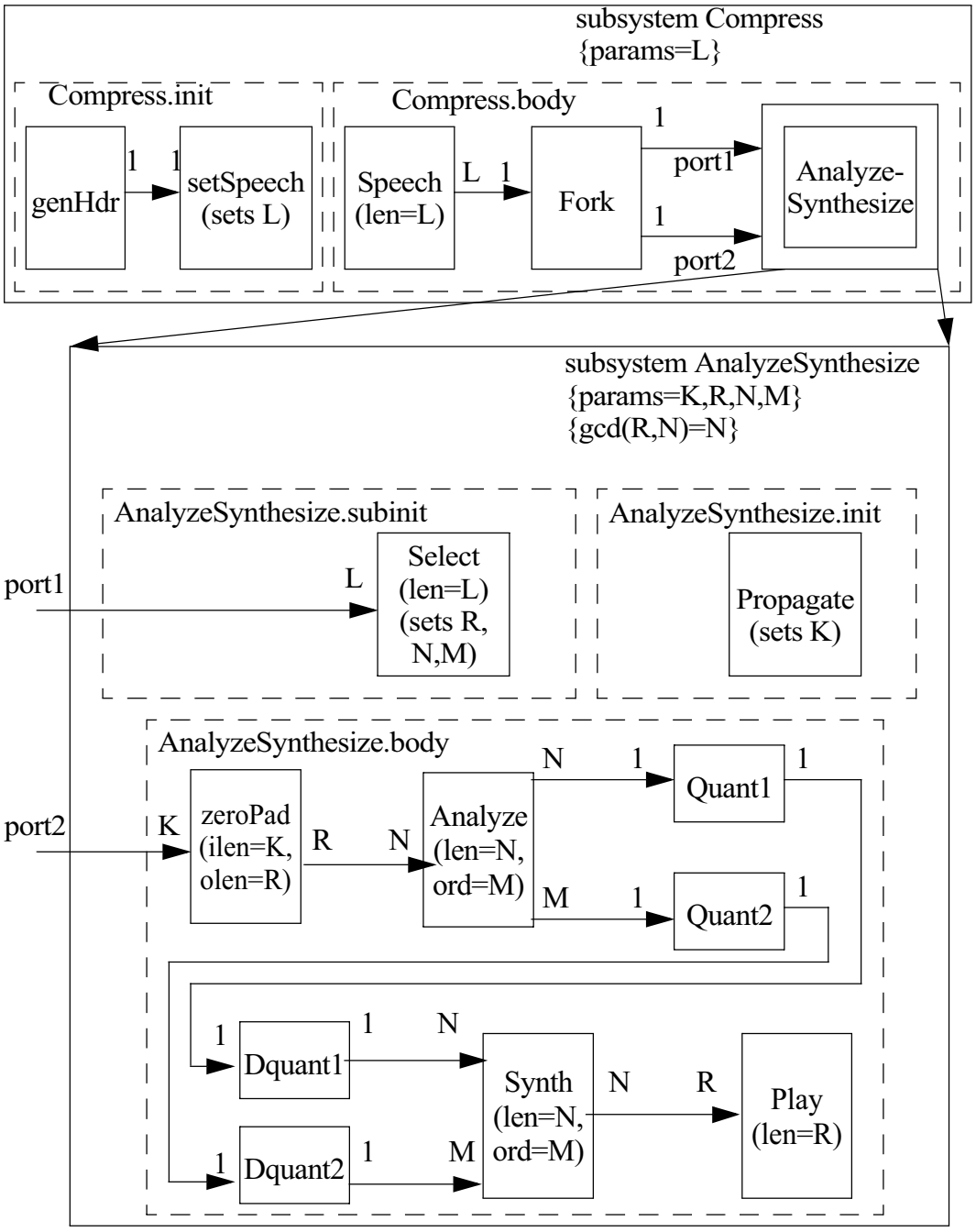


Figure 41. The Speech Compression application modeled in PSDF: *Style 2*.

the input of *zeroPad*. This is because L is an inherited parameter of *AnalyzeSynthesize*, and as specified in Section 2.1, an inherited subsystem parameter can be used in the associated subinit graph, but not in the associated body graph. Hence, the subinit graph can use L, but not the body graph. Thus, the new parameter K is necessary, which can be used in the body graph, and whose value has to be set to the inherited parameter L. The new parameter K appears on an interface edge of *AnalyzeSynthesize*, and will be visible in the parent graph (the body graph of *Compress*), and so it has to be set up as an external parameter of *AnalyzeSynthesize* in the init graph. This function is performed by the *Propagate* actor that simply assigns the value of L to K.

Fig. 42 shows the corresponding quasi-static schedule. It is very similar to the earlier schedule, but there are two points to be noted here. First, while clustering the *Fork* and *AnalyzeSynthesize* actors in the body graph of *Compress*, the scheduler has generated a sample rate consistency constraint that checks if L is equal to K. The scheduler does not know that these two parameters are equal, and with that knowledge, the need to verify this constraint at run-time would have been eliminated. Second, in the previous model (Fig. 38), there were no dataflow interfaces between graphs and their parent graphs, and thus, local synchrony verification was not an issue. In this model, however, there is dataflow communication from the body graph of *Compress* to the subinit graph and body graph of *AnalyzeSynthesize*, and thus local synchrony verification must be performed by the scheduler. For the subinit graph, it is straightforward to deduce an interface token flow of L, that is

```
repeat 5 times {
    /* begin init graph schedule for Compress */
    fire genHdr
    fire setSpeech /* sets L */
    /* end init graph schedule for Compress */

    /* begin init graph schedule for AnalyzeSynthesize */
    fire Propagate /* sets K */
    /* end init graph schedule for AnalyzeSynthesize */

    /* begin body graph schedule for Compress */
    if ((L/K) ≠ 1)
        error("sample rate inconsistent")
    fire Speech
    repeat (L) times { fire Fork }

    /* begin subinit graph schedule for AnalyzeSynthesize */
    fire Select /* sets R, N, M */
    /* end subinit graph schedule for AnalyzeSynthesize*/

    /* begin body graph schedule for AnalyzeSynthesize*/
    fire zeroPad
    repeat (R/N) times {
        fire Analyze
    }
    repeat (R) times {
        fire Quant1; fire Dquant1
    }
    repeat (M×(R/N)) times {
        fire Quant2; fire Dquant2
    }
    repeat (R/N) times {
        fire Synth
    }
    fire Play
    /* end body graph schedule for AnalyzeSynthesize */

    /* end body graph schedule for Compress */
}
```

Figure 42. The quasi-static schedule for the Speech Compression application (*Style 2*) of Fig. 41.

independent of any parameters of *AnalyzeSynthesize*. For the body graph, however, if the scheduler does not know the *gcd* relationship between R and N, then the interface token flow will be computed as being dependent on these two internal parameters of *AnalyzeSynthesize*, and hence local synchrony verification constraints will be generated. Thus, this model has a greater dependency on user assertions for the generation of an efficient quasi-static schedule, compared to the model of Fig. 38. Also, the complexity of the schedule is greater in this case, which translates to larger code size. On the other hand, generating the speech sample only once, instead of twice, per run of the application is likely to be more efficient from the perspective of execution time.

5.1.3 Zero-padding the Speech Sample

We have seen that in both of the PSDF models (Fig. 38, and 41), it is necessary to zero-pad an instance of the speech sample such that the segment size exactly divides the sample length. This necessity arises as an inherent part of the underlying SDF model. Consider an SDF design of the body graph of *Compress* in Style 1 with $R = L = 1000$, $N = 150$ and $M = 10$, and suppose that zero-padding has not been done. The APGAN schedule is

$$3\text{Speech2}(20(\text{Analyze}(150\Omega_1)(10\Omega_2)\text{Synth}))(3\text{Play}), \tag{70}$$

where $\Omega_1 = (\text{Quant1})(\text{Dquant1})$ and $\Omega_2 = (\text{Quant2})(\text{Dquant2})$. Thus an instance of the speech signal is generated thrice for every run of the application, which depending on the exact mechanism of speech production, may result at the

worst in incorrect functionality, and at the least in redundancy (inefficiency). An alternative in SDF is to model the speech source as producing a single sample on each invocation, but then the zero-padding issue manifests itself in the number of iterations of the system necessary for processing a single speech instance, with seven iterations producing 1050 samples from the speech source. Thus, it can be seen that PSDF inherits the zero padding requirement from the corresponding SDF model. This agrees with the fact that PSDF closely mimics the underlying SDF system, with the additional objective of parameterizing this SDF model such that the same design can be reused for different inputs by appropriately configuring the parameters. In this application, the necessity for zero-padding can be avoided if instead of SDF, CSDF is adopted as the underlying dataflow model. We present a PCSDF model of this application in Section 6.2.

5.2. Block Adaptive Filtering

In certain applications, like acoustic echo cancellation in teleconferencing, the necessity of frequency domain block adaptive filtering for computational efficiency is well known [21, 31, 16]. In Section 2.4, we had presented a PSDF model of a linear LMS adaptive filter. In this section we augment that by presenting a PSDF model of a time-domain block adaptive filter and the corresponding frequency-domain block adaptive filter. In a block adaptive filter, the incoming data sequence is sectioned into blocks, and applied to a block-FIR filter, one block at a time. The tap weights of the filter are held fixed over each block of data, so that adaptation of the filter proceeds on a block-by-block basis rather than on a sample-

by-sample basis as in the standard LMS adaptive filter.

5.2.1 Time-domain Block Adaptive Filtering

Fig. 43 shows a PSDF model of time-domain LMS block adaptive filtering, represented by subsystem *TDBAF*. The length of the input data (R), the size of each block (L) and the length of the FIR filter (M) are parameters of this subsystem. These parameters are configured in the init graph through the *setPars* actor. Actors *Input* and *desOutput* are data sources that produce the input data and the desired output data, respectively. L-point blocks of the input data are provided to the *blkFIR* actor, modeling block implementation of a FIR filter, which allows the efficient use of parallel processing. Each block is filtered with a fresh set of adapted coefficients received from the *wtCntrl* actor. The output of the filtering process is compared with

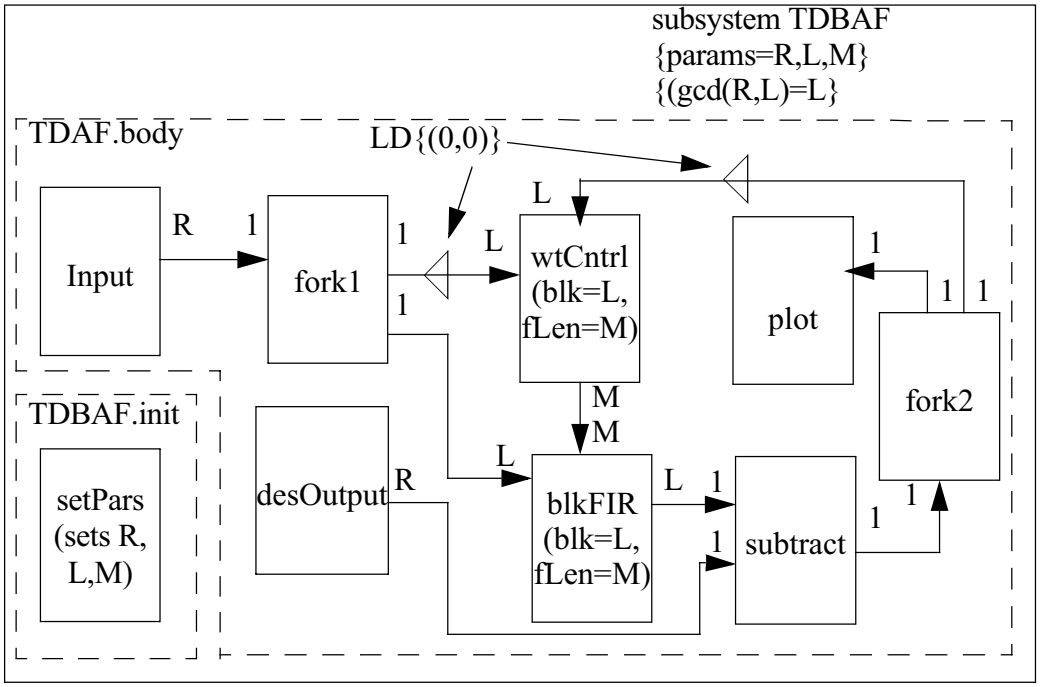


Figure 43. A PSDF model representing time-domain block adaptive filtering.

the desired output in the *subtract* actor, and the error is plotted in the *plot* actor and supplied to the *wtCntrl* actor in blocks of size L. The *wtCntrl* actor looks at the error corresponding to the most recent input block, and accordingly adapts the current filter coefficients, using the block LMS algorithm, and produces a new set of filter coefficients for processing the next input block. Both inputs of the *wtCntrl* actor are initialized with L tokens of value 0, that correspond to initiating the LMS block adaptive process with some initial values of the filter weights set in the *wtCntrl* actor. The input size and the filter length are actor parameters of both the *blkFIR* and *wtCntrl* actors, and they are assigned the subsystem parameters L and M, respectively. As before, we have omitted parameters that do not affect dataflow behavior, e.g. the step size of the weight control mechanism of the block LMS algorithm that can also be set in the init graph. In this application, we have assumed that the block size exactly divides the data size, and zero-padding as necessary is performed inside the *Input* and *desOutput* black boxes.

Our quasi-static scheduler generates the quasi-static schedule shown in Fig. 44, which implements the functionality in an obvious manner. Each run of the system processes a different set of data inputs. In each run, the *Input* and *desOutput* actors are invoked once each to produce all of the R data samples. The *fork1* actor is invoked R times to transfer the input data samples to the FIR filter and the weight control mechanism. One invocation of the *wtCntrl* actor produces a set of M coefficients, used by the *blkFIR* actor to process L input samples. After one invocation of these two actors in succession, L samples are produced at the output of the FIR,

which are then processed by the *subtract*, *fork2*, and *plot* actors in L successive invocations. To process all R data samples, this invocation sequence is then repeated (R/L) times.

5.2.2 Frequency-domain Block Adaptive Filtering

The time-domain block LMS adaptive filtering process discussed in Section 5.2.1, can be implemented in a computationally efficient manner by performing adaptation of the filter coefficients in the frequency domain using the fast Fourier transform (FFT) algorithm. The block LMS algorithm so implemented is referred to

```
repeat 5 times {
    /* begin init graph schedule for TDBAF */
    fire setPars /* sets R, L, M*/
    /* end init graph schedule for TDBAF */

    /* begin body graph schedule for TDBAF */
    fire Input
    fire desOutput
    repeat (R) times {
        fire fork1
    }
    repeat (R/L) times {
        fire wtCntrl
        fire blkFIR
        repeat (L) times {
            fire subtract
            fire fork2
            fire plot
        }
    }
    /* end body graph schedule for TDBAF */
}
```

Figure 44. The quasi-static schedule for the PSDF model of time-domain block adaptive filtering of Fig. 43.

as the *fast LMS algorithm* [21]. The linear convolution performed in the FIR filter can be speeded up by performing fast convolution using the overlap-save or overlap-add methods. It has been shown that the most efficient implementation of the fast LMS algorithm is obtained by using 50 percent overlap in the overlap-save method [14]. [21] describes a fast LMS algorithm, adapted from [31], that uses the overlap-save method with 50 percent overlap, and with the block size chosen equal to the filter length ($L = M$). A PSDF model of the fast LMS algorithm implemented in this manner is shown in Fig. 45. According to this method, the M tap weights (filter coefficients) of the FIR filter are zero-padded with an equal number of zeros, and an N-point FFT is used, where $N = 2M$. The *FDBAF* PSDF subsystem represents the frequency domain block adaptive filtering process. R, N, and M are the subsystem parameters. Fig. 45(a) shows the body graph of the *FDBAF* subsystem. The *Input* actor acts as a signal source, producing R samples of the input signal. The *Concat* block receives the input data in block sizes of M; concatenates the current block with the previous block (the kth block is appended to the $(k-1)$th block, for $k = 0, 1, \ldots, (R/M) - 1$); and provides a block of size N to the *FFT1* actor, which computes the N FFT coefficients. These coefficients are multiplied in the frequency domain with the N tap weights (computed by the weight update mechanism described shortly) in the *prod1* actor to obtain the estimated output in the frequency domain. After an inverse FFT (in *IFFT1*), we obtain N samples of the estimated output in the time domain, the first block of which is discarded by the *delFirst* actor to obtain M samples of the estimated output, corresponding to the

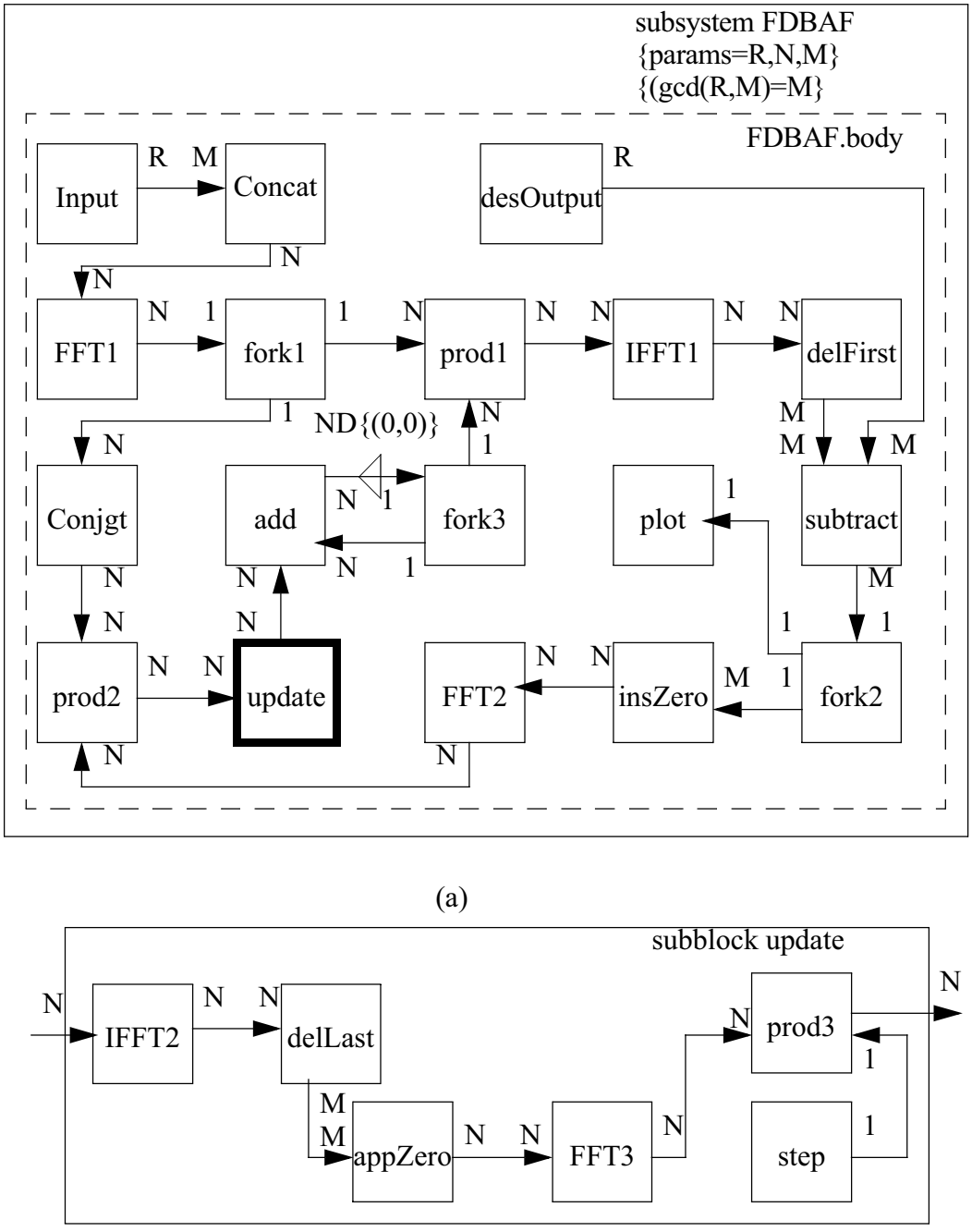


Figure 45. A PSDF model of frequency-domain block adaptive filtering.

M input samples.

For an acoustic echo cancellation application, the estimated output provides the estimated echo while the desired output comes from the actual echo [16]. The difference between the estimated output and the desired output is calculated in the *subtract* actor. The output of *subtract* provides the error signal, which determines the actual echo. The *insZero* actor inserts M zeros at the beginning of the error signal, and the *FFT2* actor transforms the error signal into the frequency domain by an FFT operation. The *Conjgt* actor computes the complex conjugate [20] of the N elements of the frequency domain input signal. The complex conjugate transpose is then multiplied in the frequency domain with the frequency domain error signal in the *prod2* actor, and provided to the *update* subblock, (enclosed by a bold rectangle in Fig. 45(a) to represent syntactic hierarchy, and shown separately in Fig. 45(b)) to compute the update for the tap weights. This is obtained by performing the inverse FFT on the product input (actor *IFFT2*), deleting the last M elements (actor *delLast*), appending M zeros to the resulting data (actor *appZero*), computing the FFT (actor *FFT3*), and multiplying with twice the LMS algorithm step size μ (in *prod3*). This update is added (in the *add* actor) to the FFT coefficients of the tap weights for the k th block ($\mathbf{w}_k$) to obtain the FFT coefficients of the next set of tap weights for the $(k+1)$ th block ($\mathbf{w}_{k+1}$). The initial set of tap weight FFT coefficients $\mathbf{w}_0$ is provided as N delay elements initialized to 0 on the output edge of the *add* actor.

In this PSDF representation, we have denoted the data flow in the system on an element-by-element basis, and it is up to the functionality of an actor to interpret

the input data appropriately (as a column vector, row vector, matrix, or identity matrix) according to the algorithm described in [21], perform the computations, and provide each element at its output. The subsystem parameters that do not affect data-flow behavior have been omitted. So as not to clutter up the block diagram, all actor parameters have also been omitted. From the block diagram, actor parameters that determine the dataflow behavior of an actor, and the subsystem parameters assigned to them can be deduced easily. For example, each of the FFT and IFFT actors have a *len* parameter denoting the FFT size. Each of these *len* parameters is assigned the subsystem parameter N. The init graph of the subsystem has not been shown. It is exactly similar to the init graph of the *TDBAF* subsystem, with a single *setPars* actor setting up all the subsystem parameters.

The quasi-static schedule is shown in Fig. 46 for five runs of the application. The block adaptive filtering process is initiated by firing the *fork3* actor N times. This transfers the N FFT coefficients of the initial tap weights to the input of the *prod1* actor, which performs a convolution product with the FFT of the input in the frequency domain. As before, we assume that the block size M, which is also equal to the filter length, divides the input length exactly. For an input signal of length R, the adaptive filtering process is applied (R/M) times, once on each block of size M. For processing a single block, the *fork1* and *fork3* actors operate on N samples, for a total of $(R/M)N = 2R$ invocations, while the *subtract*, *fork2*, and *plot* actors process M samples for a total number of $(R/M)M = R$ invocations. Every other actor in the adaptive filtering process is invoked once per filtering step, for a total

number of (R/M) invocations each. In this case, the quasi-static scheduling process is not affected by the relationship between N and M, as sample rate changes do not appear on any edge in terms of these two parameters. Thus, the schedule is not affected even though the scheduler is unaware that $N = 2M$.

5.3. CD to DAT Sample Rate Conversion

Fig. 47 shows a PSDF model of a sample rate conversion system to interface a compact disc (CD) player to a digital audio tape (DAT) player. The sample rates of

```
repeat 5 times {
    /* begin init graph schedule for FDBAF */
    fire setPars /* sets R, N, M */
    /* end init graph schedule for FDBAF */

    /* begin body graph schedule for FDBAF */
    fire Input; fire desOutput;
    repeat (R/M) times {
        fire Concat; fire FFT1; fire Conjgt;
        repeat (N) times {
            fire fork1; fire fork3;
        }
        fire prod1; fire IFFT1; fire delFirst;
        repeat (M) times {
            fire subtract; fire fork2; fire plot;
        }
        fire insZero; fire FFT2; fire prod2;
        fire IFFT2; fire delLast; fire appZero;
        fire FFT3; fire step; fire prod3;
        fire add
    }
    /* end body graph schedule for FDBAF */
}
```

Figure 46. The quasi-static schedule for the PSDF model of frequency-domain block adaptive filtering of Fig. 45.

CD players and DAT players are respectively 44.1 kHz and 48 kHz. Interfacing the two, for example, to record a CD onto a digital tape, requires a sample rate conversion. This can be done efficiently by a multi-stage implementation of the sample rate conversion [34]. Fig. 47 shows a 4-stage parameterized implementation. The sample rate conversions are performed by 4 polyphase FIR filters that respectively perform output-to-input conversion ratios i_j:d_j for $j = 1, 2, 3, 4$. Each i_j and d_j represents the interpolation and decimation factors of polyphase FIR filter j. Each of these quantities are subsystem parameters of the PSDF subsystem *CD-DAT*, and are configured in the init graph through the *setFactors* actor.

Rate conversion ratios are chosen by examining the prime factors of the two sampling rates. The ratio 441000:48000 is $3^1 7^2 : 2^5 5^1$ or 147:160. There are vari-

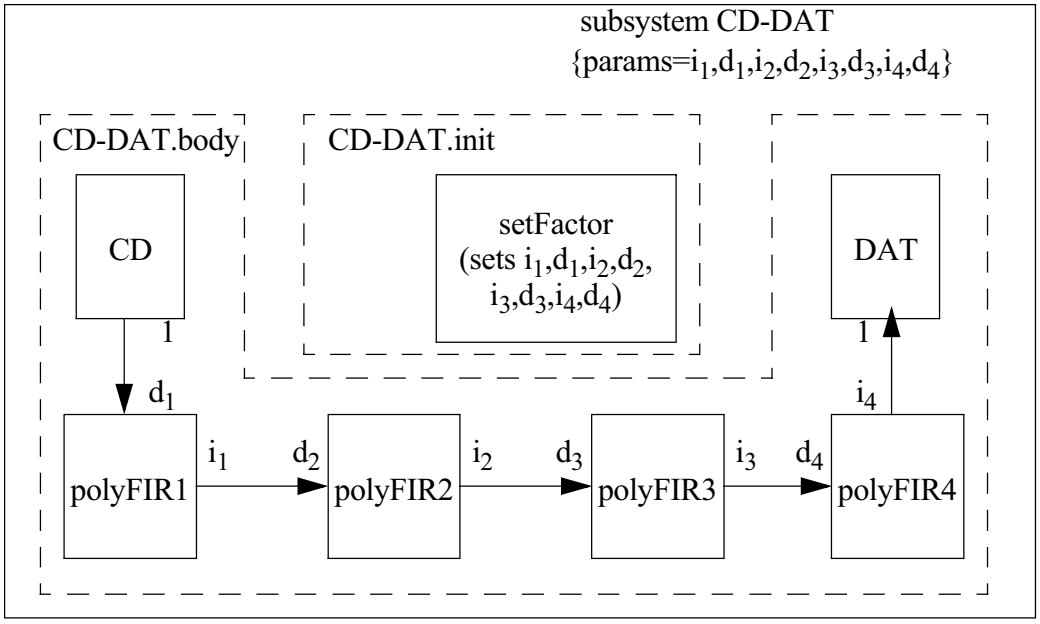


Figure 47. The PSDF model of a CD to DAT sample rate conversion system implemented in four stages with four polyphase FIR filters.

ous ways in which to perform this conversion in four stages, and the application designer can experiment with different values in different runs of the system. Two possible choices are $(2:1,\ 4:3,\ 4:7,\ 5:7)$ and $(5:3,\ 2:7,\ 4:7,\ 4:1)$.

The polyphase FIR filter *polyFIR1* will have actor parameters *decimationFactor* and *interpolationFactor*, set to the subsystem parameters d_1 and i_1 respectively. In addition it will possess some parameters that do not affect its dataflow behavior, such as the *decimationPhase* which is set to zero, and the filter coefficients which can be read from a table corresponding to permissible pairs (i_1, d_1). This is true for each of the polyphase FIR filters used in the application.

The quasi-static schedule is shown in Fig. 48. This is the general schedule for any values of the decimation and interpolation factors. To maintain fixed relationships between some of these factors, the user can additionally specify such relationships via user assertions, which may result in a more simplified schedule.

```
repeat 10 times {
    /* begin init graph schedule for CD-DAT */
    fire setFactors /* sets i1, d1, i2, d2, i3, d3, i4, d4 */
    /* end init graph schedule for CD-DAT */

    /* begin body graph schedule for CD-DAT */
    int _gcd_1 = gcd(i1, d2)
    int _gcd_2 = gcd((i2 × i1)/(_gcd_1), d3)
    int _gcd_3 = gcd((i3 × i2 × i1)/(_gcd_2 × _gcd_1), d4)
    repeat ((d4)/_gcd_3) times {
        repeat ((d3)/_gcd_2) times {
            repeat ((d2)/_gcd_1) times {
                repeat (d1) times {
                    fire CD
                }
                fire polyFIR1
            }
            repeat ((i1)/_gcd_1) times {
                fire polyFIR2
            }
        }
        repeat ((i2 × i1)/(_gcd_2 × _gcd_1)) times {
            fire polyFIR3
        }
    }
    repeat ((i3 × i2 × i1)/(_gcd_3 × _gcd_2 × _gcd_1)) times {
        fire polyFIR4
    }
    repeat (i4) times {
        fire DAT
    }
    /* end body graph schedule for CD-DAT */
}
```

Figure 48. The quasi-static schedule for the PSDF model of the 4-stage CD to DAT sample rate conversion system of Fig. 47.

Chapter 6. Extending PSDF

6.1. PSDF with Variable Topologies

The PSDF model discussed so far parameterizes the functionality and the token flow of a synchronous dataflow graph. It is possible to naturally extend the model to parameterize the graph topology also. Recall that a PSDF edge can have parameters, and the delay on the edge, along with the initial value and the re-initialization period of each delay token, can be a function of its parameter set, specified, in general, in the edge's parameter interpretation function. In a similar concept, the connectivity information of the edge, specified through its source and sink actors can also be made dependent on its parameter set. Thus, $src(e)$ and $snk(e)$ no longer must take on fixed actor port values, but can vary as determined by the parameter configuration of e. Just like the other variable quantities in a PSDF graph, the corresponding subinit or init graph is responsible for fixing up the connectivity information and configuring it as an SDF graph to be maintained for a certain number of the graph invocations. Configuring the connectivity of the PSDF graph prior to executing it is somewhat similar to the concept of *higher order functions* (HOF) present in Ptolemy [12]. HOF actors (called *stars* in Ptolemy) typically replace themselves with one or more instances of another star or *galaxy* (subblock), or they alter the connections in the graph without adding any new blocks, and then self-destruct in the pre-initialization phase of the graph, before the graph is sched-

uled. So the scheduler never encounters any HOF stars in the graph. Somewhat analogously, in PSDF, the run-time scheduler never comes across variable graph topology, as that has already been resolved and the graph has been configured as an SDF graph. However, the quasi-static scheduler does have to take variable graph topology into consideration, as we will elaborate shortly.

The BDF model and the Well Behaved stream flow model make use of two special non-SDF actors *switch* and *select* (or *merge*), which can be used effectively to express dynamic graph connectivity. Fig. 49 shows an *if-then-else* construct modeled in BDF and in PSDF. In the BDF model actor *B* emits a boolean valued token

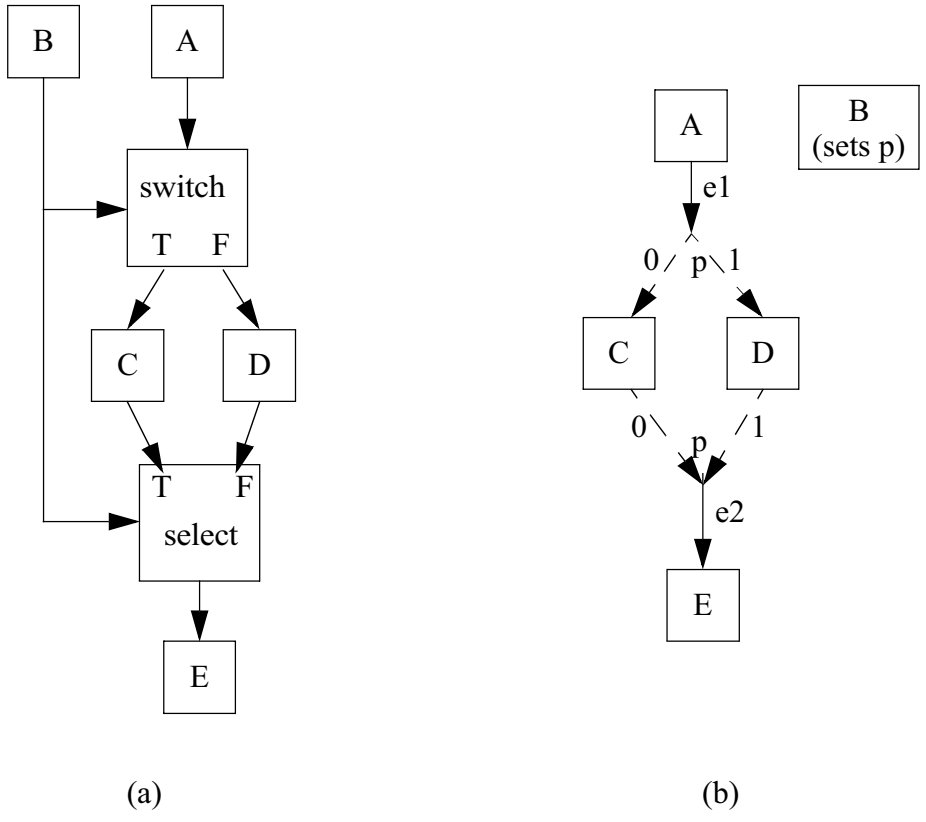


Figure 49. The *if-then-else* statement. (a) BDF representation. (b) PSDF representation.

at its output that serves as the control token for both the *switch* and the *select* actors. In the PSDF representation, dashed edges converging on an PSDF edge e indicates that $src(e)$ can vary at run-time, while dashed edges diverging from e indicates that $snk(e)$ can vary dynamically. Such an edge has two additional characteristics $srcCntrl(e)$ and $snkCntrl(e)$ that take on non-negative integer values, determined by the parameter configuration of e. These can be assigned static integers (in which case the graph degenerates to a fixed topology PSDF graph), or symbolic subsystem parameters, or can be left unspecified (in case of the latter, a parameter interpretation function must be provided to compute these, given a parameter configuration). The integer values taken on by $srcCntrl(e)$ are interpreted to configure $src(e)$ as follows. The dashed edges converging into e are enumerated from 0 to $j - 1$, where j is the total number of such dashed edges, based on some criterion (e.g., from first instantiated edge to last instantiated edge). A value of i for $srcCntrl(e)$ implies that the source actor of dashed edge i is the source actor of e. A value greater than $j - 1$ is considered invalid. The $snkCntrl(e)$ property is similarly interpreted. In Fig. 49, edge $e1$ has a variable sink, while edge $e2$ has a variable source. The property $snkCntrl(e1)$ has been assigned the subsystem parameter p while $srcCntrl(e2)$ has also been assigned the same parameter p, and actor B is responsible for configuring parameter p. The numbers on the dashed edges indicate the enumeration assigned to them. Token flow has not been specified in the figure. All the actors are assumed to be homogeneous SDF actors.

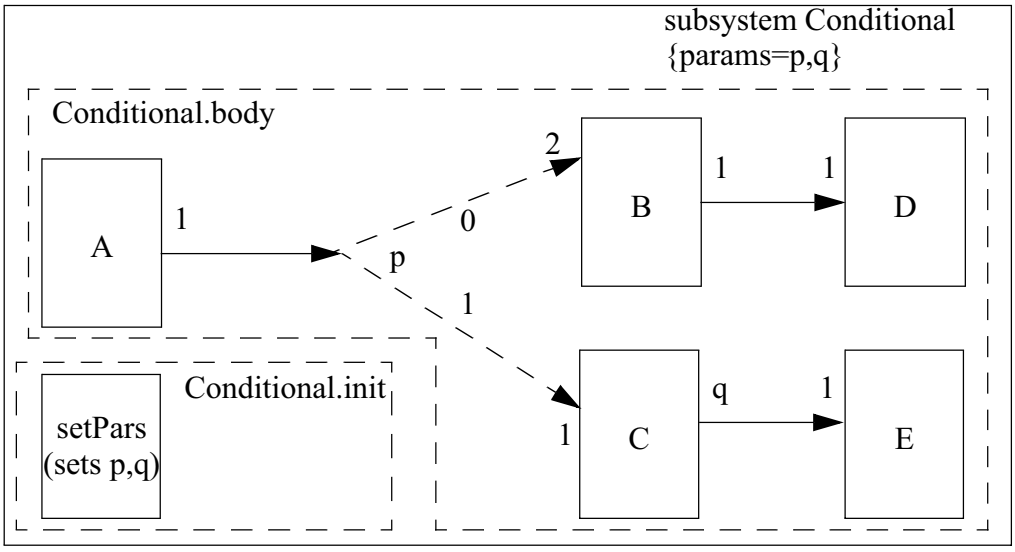
Efficient, and general quasi-static scheduling of such variable topology

PSDF graphs is a challenge. One possibility is to apply P-APGAN to each thread of execution separately, and put in the necessary *if-then-else* statements in the schedule. However, the code size explodes rapidly with the number of variable connectivity edges. Efficient management of the code size increase is a useful direction for further study. Fig. 50 shows a simple example of a PSDF specification with a single variable connectivity edge. The PSDF specification *Conditional* is shown in Fig. 50(a), where the dashed edges are annotated with their enumeration, as in Fig. 49. Fig. 50(b) specifies a possible quasi-static schedule.

6.2. Parameterization as a Meta-modeling Technique

The PSDF model applies the parameterization concept to the synchronous dataflow formalism. As discussed in Chapter 2, it should also be possible to apply the same parameterization techniques to other static models and obtain similar extensions. In fact, it should be applicable to arbitrary dataflow models by imposing a hierarchy discipline, and requiring that certain properties hold for a given subsystem over a well-defined period of time. Dataflow models that have a well-defined concept of a graph iteration are particularly amenable to parameterized extensions.

For example, cyclo-static dataflow (CSDF) can be extended to a parameterized cyclo-static dataflow (PCSDF) model, that has the same appealing re-configuration-related properties as PSDF. An illustration is given in Fig. 51 that models the speech compression application (Section 5.1) in PCSDF. This is a straightforward extension of the PSDF specification of Fig. 38 (Section 5.1.1). Recall that in the



(a)

```
repeat 5 times {
    /* begin init graph schedule for Conditional */
    fire setPars /* sets p , q */
    /* end init graph schedule for Conditional */

    /* begin body graph schedule for Conditional */
    if (p = 0) {
        repeat (2) times {
            fire A
        }
        fire B
        fire D
    } else if (p = 1) {
        fire A
        fire C;
        repeat (q ) times { fire E }
    } else
        error("invalid sink configuration")
    /* end body graph schedule for Conditional */
}
```

(b)

Figure 50. An example of a PSDF specification using a single variable connectivity edge.

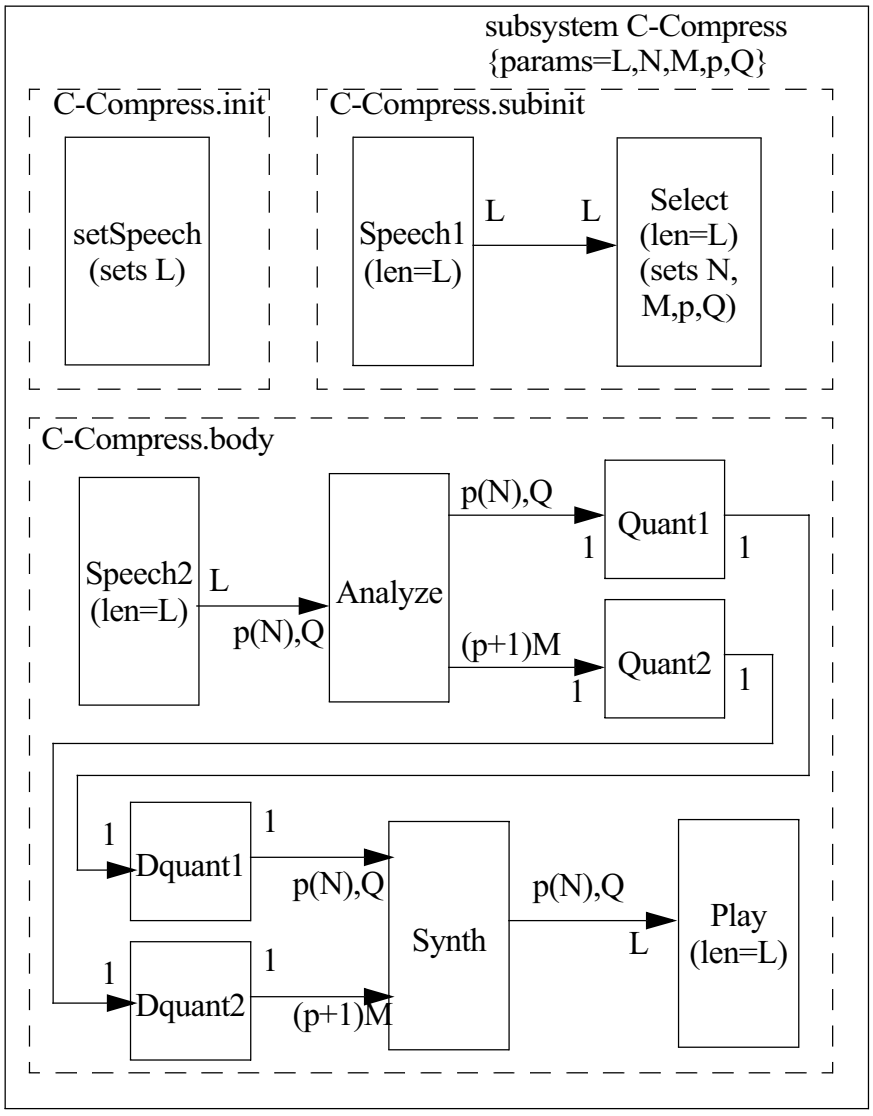


Figure 51. The PCSDF (Parameterized Cyclo-static Dataflow) specification of the speech compression application.

PSDF specification, an instance of the speech sample of length L had to be zero-padded to length R, such that the size of each segment (N) exactly divides the zero-padded length. This zero-padding is no longer necessary in the PCSDF specification. 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 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*, the code for the *Analyze* and *Synth* actors have been divided up into phases, so that their token flow can vary even in the same invocation of the parent graph, unlike PSDF, where the token flow can vary only across invocations of the parent graph. The notation $p(N), Q$ denotes the parameterized cyclo-static token flow 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 *Analyze* 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 shown in Fig. 52. The actor invocations enclosed in the first p firings process p segments of the speech sample, each of length N. The next block of actor executions process the residual segment of length Q.

In summary, we have demonstrated an informal, intuitive concept of PCSDF,

illustrated through the speech compression application. A rigorous formalism in the lines of PSDF calls for further investigation.

6.3. Summary and Future Work

The parameterized synchronous dataflow (PSDF) model significantly increases the expressivity of synchronous dataflow (SDF), by imposing a hierarchy discipline on an underlying SDF model that allows parameterized, dynamic control of a subsystem at different levels of granularity. The basic building block provided

```
/* begin body graph schedule for C-Compress */
fire Speech2
repeat (p) times {
    fire Analyze
    repeat (N) times {
        fire Quant1; fire DQuant1;
    }
    repeat (M) times {
        fire Quant2; fire Dquant2;
    }
    fire Synth
}
fire Analyze
repeat (Q) times {
    fire Quant1; fire Dquant1;
}
repeat (M) times {
    fire Quant2; fire Dquant2;
}
fire Synth
fire Play
/* end body graph schedule for C-Compress */
```

Figure 52. The quasi-static schedule for the body graph of the PCSDF specification of the speech compression application of Fig. 51.

to an application designer is a PSDF specification (subsystem) Φ, that can be decomposed into three distinct PSDF graphs — the init graph Φ_i, the subinit graph Φ_s, and the body graph Φ_b. Parameters are provide to control the functional behavior as well as the dataflow behavior (token production and consumption) of a subsystem, and the parameters can change dynamically, thus dynamically modifying system behavior. In a specification, the init and subinit graphs control the behavior of the body graph by configuring the body graph parameters appropriately. A specification can be embedded within another specification, giving rise to a powerful hierarchical structure that an application designer can use to represent nested and concurrent logical functional units capable of dynamically configuring their internal functional and dataflow behavior. 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 choice for modeling a broad class of data-dependent, dynamic DSP systems. The underlying SDF model makes the semantics intuitive and easy to understand. Furthermore, the parameterized framework provides a natural concept of re-configurability and design re-use that are desirable properties of any modeling environment. From our experience it seems that a significant population of practical, data-dependent DSP applications can be represented naturally in the PSDF model.

From the performance viewpoint, PSDF possesses a robust and elegant operational semantics that provides a promising framework for developing elaborate

verification techniques (for verifying qualities such as bounded memory execution and local synchrony), but does not rely on rigorous verification for correct operation. Efficient quasi-static schedules, geared towards minimizing code size and buffer memory requirements, can be developed for a large class of PSDF specifications, making it attractive for synthesis purposes, and optimized synthesis of quasi-static schedules appears to be an interesting area for future work. The parameterized framework of the PSDF model looks promising as naturally extensible for incorporating dynamically re-configurable *graph topologies* (conditionals), thus further increasing its expressivity. In addition, the parameterized modeling architecture that underlies PSDF appears to hold promise as a general meta-modeling technique applied to alternative dataflow models beyond SDF.

APPENDIX

In this section, we provide selected portions of the C code that we have used in the implementation of the PSDF tool (Section 4.5.2) that accepts an input PSDF specification, performs quasi-static scheduling or run-time scheduling (as appropriate), and generates a schedule for the input PSDF specification.

```
/********************* begin attr.h ****************************/
#ifndef _attr_h
#define _attr_h

#include "hashlst.h"
#include "useful.h"

enum attr_status {
  attr_status_ok,
  attr_status_out_of_range,
  attr_status_negative,
  attr_status_absent,
  attr_status_already_exists
};
/* alternate name for a pointer to a hash table */
typedef hashlst_ptr attr_list;
/* define a structure which will contain attr_list. This will contain the
 * attributes to be stored with each actor. This is the vtype that
 * will be passed to LEDA, i.e. the property of each actor. Possibly we might
 * add some other fields later.
 */
typedef struct attr_actor {
  attr_list alist;
} attr_actor;
/* Similarly define the property to be stored with each arc (etype) */
typedef struct attr_arc {
        attr_list alist;
} attr_arc;
/* Similarly define the property to be stored with each cluster */
typedef struct attr_cluster {
        attr_list alist;
} attr_cluster;
```

```
.....
boolean attr_exists(const attr_list, int);

#endif
/********************** end attr.h ****************************/

/********************** begin actor_attr.h ***********************/
#ifndef _actor_attr_h
#define _actor_attr_h

enum actor_attr {
        actor_attr_number,
        actor_attr_name,
        actor_attr_numports,
        .........
        actor_attr_psdf_repetitions,
        /* this attribute is present if the actor represents a subsystem */
        actor_attr_psdf_is_subsystem,
        /* this attribute is present if the actor is an interface node,
         * and communicates with an actor in the parent graph
         */
        actor_attr_psdf_interface_node,
        ........
};
#endif
/********************** end actor_attr.h ***********************/

/********************** begin actor_attr_numports.h **************/
#ifndef _actor_attr_numports_h
#define _actor_attr_numports_h

#include "attr.h"
#include "actor_attr.h"

attr_status actor_attr_numports_create(attr_list);
attr_status actor_attr_numports_set(attr_list, int);
attr_status actor_attr_numports_get(attr_list, int *);
attr_status actor_attr_numports_delete(attr_list);
void actor_attr_numports_print(FILE *, const void *);

#endif
/********************** end actor_attr_numports.h **************/

/********************** begin actor_attr_numports.c **************/
```

```
#include “actor_attr_numports.h”

attr_status actor_attr_numports_create(attr_list alist)
{
  void *v;
  int *key, *data;

  key = allocate_scalar(int); *key = actor_attr_numports;
  if ((v = hashlst_find_element(alist, (void *)key)) != NULL)
    return(attr_status_already_exists);
  data = allocate_scalar(int); *data = 0;
  hashlst_add_element(alist, (void *)key, (void *)data, actor_attr_numports_print);
  return(attr_status_ok);
}

attr_status actor_attr_numports_set(attr_list alist, int p)
{
  int *p1, *p2;

  p1 = allocate_scalar(int); *p1 = actor_attr_numports;
  if (p2 = (int *)hashlst_find_element(alist, (void *)p1))
  {
    *p2 = p; free(p1);
    return (attr_status_ok);
  } else {
    free(p1);
    return(attr_status_absent);
  }
}

attr_status actor_attr_numports_get(attr_list alist, int *p)
{
  int *p1, *p2;

  p1 = allocate_scalar(int); *p1 = actor_attr_numports;
  if (p2 = (int *)hashlst_find_element(alist, (void *)p1))
  {
    *p = *p2; free(p1);
    return (attr_status_ok);
  } else {
    free(p1);
    return(attr_status_absent);
  }
}
```

```
attr_status actor_attr_numports_delete(attr_list alist)
{
 int *p1, *p2;

 p1 = allocate_scalar(int); *p1 = actor_attr_numports;
 if ((hashlst_del_element(alist, (void *)p1)) == FALSE)   {
  free(p1);
  return(attr_status_ok);
 } else {
  free(p1);
  return(attr_status_absent);
 }
}

void actor_attr_numports_print(FILE *fp, const void *data)
{
 int d = (*(int *)data);
 fprintf(fp, "%d ", d);
}
```

/********** **end actor_attr_numports.c** **********/

/********** **begin psdf_sched.c** **********/

```
#include <string.h>
#include "error.h"
#include "psdf_read_graph.h"
#include "cluster.h"
#include "psdf_cluster.h"
#include "psdf_schedule.h"
#include "psdf_run_time_schedule.h"

/* global variable to indicate whether or not quasi-static
 * scheduling is done, written in psdf_read_graph.c
 */
boolean G_quasi_static;
int main(int argc, char **argv) {

 /* local variable declarations */

 FILE *psdf_file, *param_file, *sched;
 leda_actor_GRAPH_ptr g;
 leda_cluster_GRAPH_ptr c;
 int num_firings = 1;
```

```
  sched = fopen("output.sched", "w");
  G_quasi_static = TRUE;
  if (argc == 2) {
  /* format - read_graph <filename.psdf> */
    psdf_file = open_psdf_file(argv[1]);
  } else if (argc == 3) {
  /* format "read_graph <filename> {<num-firings>}" */
    psdf_file = open_psdf_file(argv[1]);
    num_firings = atoi(argv[2]);
  } else if (argc == 4) {
  /* format "read_graph <filename> {<num-firings>} {<quasi-static/run-time>}" */
    psdf_file = open_psdf_file(argv[1]);
    num_firings = atoi(argv[2]);
    G_quasi_static = atoi(argv[3]);
  } else
        error("usage: psdf_sched <filename.psdf> {<num-firings>} {<static/run-
time>}");
  if (!psdf_file)
     error_fatal("Could not open file %s", argv[1]);

  g = psdf_read_graph(psdf_file);
  /* schedule according to QSS or run-time */
  if (G_quasi_static) {
    /* QSS scheduling */
    printf("Doing quasi-static scheduling\n");
    /* initialize cluster hierarchy */
    c = new leda_cluster_GRAPH;
    cluster_init_hierarchy(c, g);
    psdf_cluster_init_hierarchy(c);
    psdf_schedule_graph(c);
    psdf_cluster_print_schedule(sched, c, num_firings);
  } else {
    /* run-time scheduling */
    printf("Doing run-time scheduling\n");
    param_file = open_param_file(argv[1]);
    if (!param_file) {
      error_warn(".param file not provided\n");
      param_file = NULL;
    }
    psdf_run_time_schedule_graph(param_file, sched, g, num_firings);
  }
  fclose(sched);
}
```

```
/*************** end psdf_sched.c ***************************/

/*************** begin psdf_schedule.c ***********************/
/* #include files and local function declarations */
.....
void psdf_schedule_graph(leda_cluster_GRAPH_ptr top_graph)
{
  leda_node supernode;

  /* the topmost cluster is a dummy. It stays as the container of a single
   * supernode, which points to the subgraph that the user has specified as
   * the topmost graph. Extract that supernode.
   */
  supernode = top_graph->first_node();
  psdf_schedule_supernode(top_graph, supernode);
  return;
}

void psdf_schedule_supernode(leda_cluster_GRAPH_ptr supergraph,
                             leda_node supernode)
{
  attr_cluster su_attr, node_attr;
  attr_cluster_ptr su_attr_ptr;
  attr_actor ac_attr;
  leda_cluster_GRAPH_ptr *graph_ptr;
  leda_cluster_GRAPH_ptr my_body_graph, my_init_graph, my_subinit_graph;
  leda_node control_cluster;
  psdf_input_interface_ptr input;
  psdf_polynomial_ptr cn, snk_cn, cn_new, pr, pr_new, l_constr;
  psdf_polynomial_list_ptr init_constr_list, subinit_constr_list;
  boolean locally_synchronous;
  leda_edge inp_edge, otp_edge;
  attr_arc arc_attr;
  int param_value;
  char su_name[MAXNAMELEN];
  psdf_parameter_list_ptr *param_list, p;
  psdf_cluster_interface_list_ptr *cl_intf_ptr;

  su_attr = supergraph->inf(supernode);
  graph_ptr = allocate_scalar(leda_cluster_GRAPH_ptr);
  su_attr_ptr = allocate_scalar(attr_cluster);
  *su_attr_ptr = su_attr;
  init_constr_list = allocate_scalar(psdf_polynomial_list);
  subinit_constr_list = allocate_scalar(psdf_polynomial_list);
```

```
/* obtain the attributes of the hierarchical actor that is contained
 * inside the supernode cluster in the variable ac_attr
 */
.........
actor_attr_name_get(ac_attr.alist, su_name);

/* Schedule the init graph. First schedule
 * child subsystems, then schedule the init graph
 */

if (attr_exists(su_attr.alist, cluster_attr_psdf_init_graph)) {
  cluster_attr_psdf_init_graph_get(su_attr.alist, graph_ptr);
  my_init_graph = *graph_ptr;
  forall_nodes(nd, *my_init_graph) {
    node_attr = my_init_graph->inf(nd);
    if (attr_exists(node_attr.alist, cluster_attr_psdf_is_subsystem)) {
      psdf_schedule_supernode(my_init_graph, nd);
    }
  }
  psdf_schedule_subgraph_APGAN(my_init_graph, su_attr_ptr);
  /* recurse through the schedule hierarchy computing the product
   * of the repetition count and tokens produced at each output port of
   * each leaf actor. Check if a leaf actor configures a parameter at an
   * output port, then that product should be 1. If can't verify at
   * compile-time, then put in constraint
   */
  control_cluster = my_init_graph->first_node();
  psdf_check_unit_transfer(my_init_graph, control_cluster,
                    psdf_polynomial_allocate_from_int(1),
                    init_constr_list);
  psdf_process_constraint_list(su_attr_ptr, init_constr_list);
}

/* schedule subinit graph similarly */
......

/* Now schedule the body graph. Schedule child subsystems first.
 */

 if (attr_exists(su_attr.alist, cluster_attr_psdf_body_graph)) {
  cluster_attr_psdf_body_graph_get(su_attr.alist, graph_ptr);
  my_body_graph = *graph_ptr;
  forall_nodes(nd, *my_body_graph) {
```

```
    node_attr = my_body_graph->inf(nd);
    if (attr_exists(node_attr.alist, cluster_attr_psdf_is_subsystem)) {
      psdf_schedule_supernode(my_body_graph, nd);
    }
  }
  /* all child subgraphs of the body graph have been successfully
   * scheduled. Now schedule the body graph
   */
  psdf_schedule_subgraph_APGAN(my_body_graph, su_attr_ptr);
}

/* look at each interface token flow expression. If it does not depend on
 * non-init-configured parameters then no need of run-time constraint checking.
 * Else, evaluate each expression with default values for non-init-configured
 * parameters. Assign that as token flow of parent edge. Insert a constraint
 * for checking this evaluated expression with actual expression. e.g. for
 * expression 2*(p)*(n/m), where p is init-set and n, m are non-init set params
 * with default(n)=4, default(m)=2, then constraint will be
 * "if ((2*(p)*2) != (2*(p)*(n/m)), error("inconsistent graph")". This will
 * be put in after subinit schedule and after body schedule.
 */

cl_intf_ptr = allocate_scalar(psdf_cluster_interface_list_ptr);
cluster_attr_psdf_interface_get(su_attr_ptr->alist, cl_intf_ptr);
if (*cl_intf_ptr) {
  param_list = allocate_scalar(psdf_parameter_list_ptr);
  actor_attr_psdf_parameters_get(ac_attr.alist, param_list);

  /* process interface input edges */

  for (input = (*cl_intf_ptr)->input; input; input = input->next) {
    cn = input->sink_consumed;
    cn_new = psdf_polynomial_product(cn,
                                psdf_polynomial_allocate_from_int(1));
    locally_synchronous = TRUE;
    inp_edge = *(input->input_edge);
    arc_attr = supergraph->inf(inp_edge);
    arc_attr_psdf_tokens_consumed_set(arc_attr.alist, cn);
    supergraph->assign(inp_edge, arc_attr);
    /* check if this is a function of a supernode parameter */
    if (!(*param_list)) {
    /* no parameters, hence cn should be a pure number */
      if (psdf_polynomial_is_pure_number(cn_new) <= 0)
          error_fatal("psdf_schedule_supernode: tokens consumed by subgraph %s
```

```
should be a positive integer", su_name);
      } else {
      /* check if cn is a function of any parameter */
        for (p = *param_list; p; p = p->next) {
          if (psdf_polynomial_is_function_of_polynomial(cn_new, p->name)) {
            if ((p->param_type == psdf_param_type_internal) ||
                (p->param_type == psdf_param_type_external_nondataflow)) {
              /* cannot say at compile-time if locally synchronous */
              locally_synchronous = FALSE;
              /* evaluate the token flow with the default value of this parameter */
              if (p->data_type != psdf_param_data_type_int)
                error("psdf_schedule: only integer parameters can appear in token flow");
              param_value = atoi(p->default_value);
              cn_new = psdf_polynomial_evaluate_expression(cn_new,
                                                  p->name, param_value);
            }
          }
        }
      }
      if (!locally_synchronous) {
        /* assign cn_new as the consumed quantity of edge in parent graph */
        arc_attr_psdf_tokens_consumed_set(arc_attr.alist, cn_new);
        supergraph->assign(inp_edge, arc_attr);
        /* insert local synchrony constraint. if (cn_new != cn) then error,
         * which is equivalent to "if ((cn_new/cn) != 1) then error
         */
        l_constr = psdf_polynomial_divide(cn, cn_new);
        psdf_process_constraint(su_attr_ptr, l_constr);
      }
    }
    /* process interface output edges similarly */
    ......
  }
  supergraph->assign(supernode, *su_attr_ptr);
  return;
}

void psdf_schedule_subgraph_APGAN(leda_cluster_GRAPH_ptr my_graph,
                                  attr_cluster_ptr supernode_attr_ptr)
{
  leda_edge e, cluster_edge = NULL;
  leda_node v, src, snk, cluster_src = NULL, cluster_snk = NULL, n1, n2;
  boolean cycle;
```

```
/* first cluster the single-rate sub-systems, then the sdf arcs, then those
 * arcs whose gcd is known (i.e. a variable has not been assigned)
 */
psdf_schedule_single_rate(my_graph, supernode_attr_ptr);
psdf_schedule_sdf(my_graph, supernode_attr_ptr);
psdf_schedule_gcd_known(my_graph, supernode_attr_ptr);

/* now schedule the - multi-rate, psdf, gcd unknown - portion of the dataflow graph
 */
while ((my_graph->number_of_edges()) > 0) {
  /* find out a suitable cluster edge */
  forall_edges(e, *my_graph) {
    src = source(e);
    snk = target(e);
    /* check if clustering e will introduce a cycle */
    cycle = FALSE;
    forall_nodes(v, *my_graph) {
      /* clustering e will introduce a cycle if some node that is reachable
       * from src, can reach snk
       */
      if (leda_is_reachable(src, v) && leda_is_reachable(v, snk)) {
        cycle = TRUE;
        break;
      }
    }
    if (!cycle) {
      /* at this point e is valid cluster edge */
      cluster_edge = e;
      cluster_src = src;
      cluster_snk = snk;
      break;
    }
  }
  psdf_pairwise_cluster(my_graph, supernode_attr_ptr,
                        cluster_src, cluster_snk, cluster_edge);
}

/* the graph contains no more edges, now cluster the remaining nodes */
while ((my_graph->number_of_nodes()) > 1) {
  n1 = my_graph->first_node();
  n2 = my_graph->last_node();
  psdf_pairwise_cluster(my_graph, supernode_attr_ptr, n1, n2, NULL);
}
return;
```

```
}
.......

/*************** end psdf_schedule.c ***********************/

/*************** begin psdf_cluster.c ***********************/
/* #include files and local function declarations */
........
leda_node psdf_pairwise_cluster(leda_cluster_GRAPH_ptr cluster_graph,
                        attr_cluster_ptr supernode_attr_ptr,
                        leda_node cluster_source,
                        leda_node cluster_sink,
                        leda_edge cluster_edge)
{
 /* performs pairwise APGAN clustering. If cluster_edge is NULL, implies that
  * we are trying to cluster 2 disconnected components. In that case, just
  * absorb cluster_source, and cluster_sink in the new cluster, with a
  * repetition of 1 each. The new cluster won't have a gcd term, or constraints
  * term.
  */

 attr_cluster_ptr newc_attr;
 attr_cluster src_attr, snk_attr, supernode_attr;
 attr_arc cedge_attr, edge_attr;
 leda_node new_cluster;
 leda_edge other_edge, in_edge, out_edge;
 psdf_polynomial_ptr *prd_p, *cns_p, reps, reps_src, reps_snk, new_prd, new_cns;
 psdf_polynomial_ptr prd, cns, other_prd, other_cns, gc_polynomial, eqn;
 leda_cluster_GRAPH_ptr *sugraph;
 psdf_user_gcd_list_ptr user_gcd = NULL;
 psdf_polynomial_product_table_ptr user_product = NULL;

 /* The user may have provided some gcd information and some product informa-
tion
  * via user assertions. Obtain these in the variables user_gcd, and user_product
  */
 ....

 /* create a new non-leaf cluster attribute and initialize it */
 newc_attr = allocate_scalar(attr_cluster);
 ....

 /* check if cluster_edge is NULL */
 if (!cluster_edge) {
```

```
/* reps_src = reps_snk = 1; absorb the 2 clusters inside the new
 * cluster, and redirect edges to/from the new cluster. Hide src, and snk.
 * The code here is similar to that when cluster_edge is non null, but
 * simpler
 */
 ......
 /* create a new non-leaf cluster in cluster_graph with attribute newc_attr*/
 ......
 return new_cluster;
}

/* at this point cluster_edge is non NULL. Obtain tokens produced onto
 * and consumed from cluster_edge
 */

prd_p = allocate_scalar(psdf_polynomial_ptr);
cns_p = allocate_scalar(psdf_polynomial_ptr);
cedge_attr = cluster_graph->inf(cluster_edge);
arc_attr_psdf_tokens_consumed_get(cedge_attr.alist, cns_p);
arc_attr_psdf_tokens_produced_get(cedge_attr.alist, prd_p);
prd = *prd_p;
cns = *cns_p;

/* check for sample rate consistency, and build up the list of sample
 * rate consistency constraints, store it in the new cluster attributes
 */
forall_out_edges(other_edge, cluster_source) {
/* look at the other edges going between cluster_source, and cluster_sink,
 * besides cluster_edge
 */
  if (other_edge == cluster_edge)
    continue;
  if (target(other_edge) == cluster_sink) {
  /* For consistency the ratio of the # tokens produced on other_edge, to the
   * # tokens consumed from other_edge, should be the same as the
   * corresponding ratio for cluster_edge
   */
    prd_p = allocate_scalar(psdf_polynomial_ptr);
    cns_p = allocate_scalar(psdf_polynomial_ptr);
    edge_attr = cluster_graph->inf(other_edge);
    arc_attr_psdf_tokens_consumed_get(edge_attr.alist, cns_p);
    arc_attr_psdf_tokens_produced_get(edge_attr.alist, prd_p);
    other_prd = *prd_p;
    other_cns = *cns_p;
```

```
    eqn = psdf_polynomial_divide(
          psdf_polynomial_product(other_prd, cns),
          psdf_polynomial_product(other_cns, prd)
          );
    psdf_polynomial_simplify(eqn);
    if (psdf_polynomial_is_pure_number(eqn) >= 0) {
      if (!psdf_polynomial_is_equal_to_int(eqn, 1)) {
        error("psdf_pairwise_cluster: inconsistent graph");
      }
    } else {
    /* have to check at run-time */
      psdf_process_sample_rate_constr(newc_attr, eqn);
    /* hide this edge, can remove it also */
    cluster_graph->hide_edge(other_edge);
  }
}
cluster_graph->hide_edge(cluster_edge);
/* compute the gcd of the produced, and consumed terms of cluster_edge,
 * store it in new cluster attributes. Check if the gcd has already been
 * defined by the user
 */
gc_polynomial = psdf_polynomial_compute_gcd(prd, cns,
                                    user_gcd, user_product);
if (psdf_polynomial_is_gcd_function(gc_polynomial)) {
/* create a gcd term for new cluster */
  .....
}
/* compute the repetitions of cluster_source, and store it in it's
 * cluster_attr_psdf_repetitions. Do the same for cluster_sink.
 */
reps_src = psdf_polynomial_divide(cns, gc_polynomial);
reps_snk = psdf_polynomial_divide(prd, gc_polynomial);
src_attr = cluster_graph->inf(cluster_source);
snk_attr = cluster_graph->inf(cluster_sink);
cluster_attr_psdf_repetitions_set(src_attr.alist, reps_src);
cluster_attr_psdf_repetitions_set(snk_attr.alist, reps_snk);
cluster_graph->assign(cluster_source, src_attr);
cluster_graph->assign(cluster_sink, snk_attr);
/* absorb cluster_src, and cluster_snk within the new cluster */
cluster_attr_num_members_set(newc_attr->alist, 2);
......
/* create a new non-leaf cluster in cluster_graph with attribute newc_attr */
new_cluster = cluster_graph->new_node(*newc_attr);
```

```
  /* adjust the token flow of the adjacent edges to cluster_sink, and
   * cluster_source.  Redirect these edges to/from the new cluster
   */
  forall_in_edges(in_edge, cluster_source) {
    edge_attr = cluster_graph->inf(in_edge);
    cns_p = allocate_scalar(psdf_polynomial_ptr);
    arc_attr_psdf_tokens_consumed_get(edge_attr.alist, cns_p);
    cns = *cns_p;
    new_cns = psdf_polynomial_product(cns, reps_src);
    arc_attr_psdf_tokens_consumed_set(edge_attr.alist, new_cns);
    cluster_graph->move_edge(in_edge, source(in_edge), new_cluster);
  }
  forall_out_edges(out_edge, cluster_source) {
    edge_attr = cluster_graph->inf(out_edge);
    prd_p = allocate_scalar(psdf_polynomial_ptr);
    arc_attr_psdf_tokens_produced_get(edge_attr.alist, prd_p);
    prd = *prd_p;
    new_prd = psdf_polynomial_product(prd, reps_src);
    arc_attr_psdf_tokens_produced_set(edge_attr.alist, new_prd);
    cluster_graph->move_edge(out_edge, new_cluster, target(out_edge));
  }

  /* process input and output edges of cluster_snk similarly */
  ......

  /* check if the cluster_src, or cluster_snk is an interface node,
   * in that case, adjust the token flow of edges to/from them, in the
   * supernode attribute interface information
   */
  if (attr_exists(src_attr.alist, cluster_attr_psdf_interface_node))
      adjust_interface_token_flow(supernode_attr_ptr, cluster_graph, cluster_source,
new_cluster);
  if (attr_exists(snk_attr.alist, cluster_attr_psdf_interface_node))
        adjust_interface_token_flow(supernode_attr_ptr, cluster_graph, cluster_sink,
new_cluster);

  /* Remove cluster_source, and cluster_sink from cluster_graph.
   * We are not deleting the node, but hiding it. new_cluster has an attribute
   * points to cluster_source, and cluster_sink. So removing these, might
   * make those pointers invalid.
   */
   cluster_graph->hide_node(cluster_source);
   cluster_graph->hide_node(cluster_sink);
   return new_cluster;
```

```
}

void psdf_cluster_print_schedule(FILE *fp,
                        leda_cluster_GRAPH_ptr cluster_graph,
                        int num_firings)
{
 leda_node supernode;
 attr_cluster su_attr;
 psdf_polynomial_ptr *reps;

 /* the topmost cluster is a dummy. It stays as the container of a single
  * supernode, which points to the subgraph that the user has specified as
  * the topmost graph. Extract that supernode. The user specified number of
  * repetitions is stored in the top cluster. Print that if necessary.
  */
  if (num_firings == 1) {
  /* no need of printing "repeat ..." */
    fprintf(fp, "{\n");
    psdf_cluster_print_graph_schedule(fp, cluster_graph, NULL, 0);
    fprintf(fp, "}\n");
  } else {
  /* print "repeat ..." */
    fprintf(fp, "repeat ");
    fprintf(fp, "%d", num_firings);
    fprintf(fp, " times {\n");
    psdf_cluster_print_graph_schedule(fp, cluster_graph, NULL, 0);
    fprintf(fp, "}\n");
 }
}

void psdf_cluster_print_graph_schedule(FILE *fp,
                        leda_cluster_GRAPH_ptr cluster_graph,
                        psdf_polynomial_list_ptr unit_constr_list,
                        int depth)
{
 int num_nodes;
 leda_node cluster_node;

 /* this will traverse the cluster hierarchy present in the cluster of this
  * graph, and for each supernode, it will print the schedule of the init graph
  * of that supernode. After all supernodes are traversed, it will print the
  * schedule of this graph
  */
 num_nodes = (cluster_graph)->number_of_nodes();
```

```
  if (num_nodes > 1)
    error("psdf_cluster_print_supernode: improper clustering detected in dataflow
graph");
  cluster_node = cluster_graph->first_node();
  /* print the init graph schedules of all sub-systems present in this graph */
  psdf_print_init_schedule(fp, cluster_graph, cluster_node, depth+1);
  /* then print this graph's schedule as the preamble followed by the body */
  psdf_print_APGAN_schedule(fp, cluster_graph, cluster_node,
                        unit_constr_list, depth+1);
}
.....
void psdf_print_APGAN_schedule(FILE *fp,
                        leda_cluster_GRAPH_ptr cluster_graph,
                        leda_node cluster,
                        psdf_polynomial_list_ptr unit_constr_list,
                        int depth)
{
  /* print the preamble - unknown token flow declaration, gcd declaration,
   * checking unit transfer consistency. Sample rate consistency checks
   * are implemented as code with each cluster pair. Conceptually can do
   * those checks here also. After the preamble print the looped schedule
   * comprising the body.
   */

  psdf_polynomial_list_ptr *constraint, cn;
  psdf_polynomial_ptr p;
  attr_cluster cluster_attr = cluster_graph->inf(cluster);
  int new_depth = depth;

  /* print the gcd assignments, preceded by a starting brace, to specify
   * the scope of the gcd variable declarations. The closing brace comes after
   * printing the dataflow schedule, as the gcd assignments are valid up to
   * that point
   */
  if (attr_exists(cluster_attr.alist, cluster_attr_psdf_gcd)) {
    print_tab(fp, depth);
    fprintf(fp, "/* gcd variable declarations */\n");
    /* print gcds */
    psdf_print_gcd(fp, cluster_graph, cluster, depth);
  }
  /* print unit_constr_list */
  if (unit_constr_list) {
  /* print "if (contsraint) != 1, error" */
    .....
```

```
  }
  /* print looped schedule for graph */
  print_tab(fp, new_depth);
  fprintf(fp, "/* looped schedule */\n");
  psdf_print_cluster_reps(fp, cluster_graph, cluster, new_depth);
}

void psdf_print_cluster_reps(FILE *fp,
                        leda_cluster_GRAPH_ptr cluster_graph,
                        leda_node cluster,
                        int depth)
{
  psdf_polynomial_ptr *reps = allocate_scalar(psdf_polynomial_ptr);
  attr_cluster cluster_attr = cluster_graph->inf(cluster);
  cluster_members_list_ptr *members_list;
  member_list_ptr mem;
  leda_node mem_node;
  int new_depth, *leaf = allocate_scalar(int);

  new_depth = depth;
  cluster_attr_psdf_repetitions_get(cluster_attr.alist, reps);
  if (!psdf_polynomial_is_equal_to_int(*reps, 1)) {
  /* print "repeat ... {" */
    new_depth = depth+1;
    print_tab(fp, depth);
    fprintf(fp, "repeat ");
    psdf_polynomial_print(fp, *reps);
    fprintf(fp, " times {\n");
  }
  /* check if this is a non-leaf, then recursively traverse children
   */
  cluster_attr_isleaf_get(cluster_attr.alist, leaf);
  if (*leaf) {
  /* leaf node, either actor, or has a subgraph pointer */
    if (attr_exists(cluster_attr.alist, cluster_attr_psdf_is_subsystem)) {
      /* traverse the subgraph */
      psdf_cluster_print_supernode(fp, cluster_graph, cluster, new_depth);
    } else {
    /* print "fire actor" */
      .....
    }
  } else {
  /* non-leaf, recursively traverse children, check for sample rate constraints */
    if (attr_exists(cluster_attr.alist, cluster_attr_psdf_sample_rate_constr)) {
```

```
    /* print "if (contsraint) != 1, error" */
      .....
    }
    members_list = allocate_scalar(cluster_members_list_ptr);
    cluster_attr_members_list_get(cluster_attr.alist, members_list);
    for (mem = (*members_list)->members; mem; mem = mem->next) {
      mem_node = *(mem->node);
      psdf_print_cluster_reps(fp, cluster_graph, mem_node, new_depth);
    }
  }
  if (new_depth == (depth+1)) {
    print_tab(fp, depth);
    fprintf(fp, "}\n");
  }
}

void psdf_cluster_print_supernode(FILE *fp,
                        leda_cluster_GRAPH_ptr cluster_graph,
                        leda_node supernode,
                        int depth)
{
  attr_cluster su_attr;
  leda_cluster_GRAPH_ptr *subgr, cntrl_subgr, data_subgr;
  char name[MAXNAMELEN];
  psdf_polynomial_list_ptr unit_reps;

  su_attr = cluster_graph->inf(supernode);
  /* obtain the name of the subsystem represented by supernode in
   * variable name
   */
  ......
  print_tab(fp, depth);
  fprintf(fp, "{\n");
  subgr = allocate_scalar(leda_cluster_GRAPH_ptr);
  /* print schedule for subinit graph */
  if (attr_exists(su_attr.alist, cluster_attr_psdf_subinit_graph)) {
    cluster_attr_psdf_subinit_graph_get(su_attr.alist, subgr);
    cntrl_subgr = *subgr;
    print_tab(fp, depth+1);
    fprintf(fp, "/* begin subinit graph schedule for %s */\n", name);
    /* extract unit transfer subinit graph constraint from su_attr in the
     * variable unit_reps
     */
    .....
```

```
    psdf_cluster_print_graph_schedule(fp, cntrl_subgr, unit_reps, depth+1);
    print_tab(fp, depth+1);
    fprintf(fp, "/* end subinit graph schedule for %s */\n", name);
  }
  /* print schedule for body graph */
  if (attr_exists(su_attr.alist, cluster_attr_psdf_body_graph)) {
    cluster_attr_psdf_body_graph_get(su_attr.alist, subgr);
    data_subgr = *subgr;
    print_tab(fp, depth+1);
    fprintf(fp, "/* begin body graph schedule for %s */\n", name);
    psdf_cluster_print_graph_schedule(fp, data_subgr, NULL, depth+1);
    print_tab(fp, depth+1);
    fprintf(fp, "/* end body graph schedule for %s */\n", name);
  }
  /* print local synchrony verification constraints, if any */
  if (attr_exists(su_attr.alist, cluster_attr_psdf_local_synchrony_constr)) {
  /* print "if (contsraint) != 1, error" */
    ......
  }
  print_tab(fp, depth);
  fprintf(fp, "}\n");
}
.............
/*************** end psdf_cluster.c ***********************/

/*************** begin psdf_run_time_schedule.c ***************/
/* # include files and local function declarations */
......
void psdf_run_time_schedule_graph(FILE *user_file,
                                  FILE *fp,
                                  leda_actor_GRAPH_ptr gr,
                                  int num_firings)
{
  hashlst_ptr param_table, user_table;
  int i;

  /* the user can provide parameter values in a file "user_file". Read
   * that file and store the information in a hash table user_table. This
   * will have the parameter name (as a psdf_polynomial) in the KEY and the
   * data will be an array of values that the user has provided for the
   * parameter. Values will be read from user_table, and entries made into
   * the param_table, which represents the dynamic parameter setting at
   * run-time.
   */
```

```
  user_table = psdf_run_read_user_file(user_file);
  param_table = hashlst_allocate(PARAM_HASH_SIZE,
                                 polynomial_hash_func,
                                 polynomial_cmp_func,
                                 polynomial_prnt_key
                                 );
  hashlst_init(param_table);
  /* param_table will contain {<parameter-name>, <value>}  tuples
   * hashed by parameter-name. This table will be filled in
   * by init and subinit graphs. When exiting a sub-system,
   * parameter entries for that sub-system will be deleted
   * from the table
   */
  for (i = 0; i < num_firings; i++) {
    fprintf(fp, "Firing %d\n", i+1);
    fprintf(fp, "{\n");
    psdf_run_sched_graph(fp, gr, user_table, param_table);
    fprintf(fp, "}\n");
  }
  return;
}

void psdf_run_sched_graph(FILE *fp,
                          leda_actor_GRAPH_ptr gr,
                          hashlst_ptr user_table,
                          hashlst_ptr param_table)
{
  leda_node nd, act;
  attr_actor nd_attr;
  char actor_name[100];
  leda_actor_GRAPH_ptr *subgr, init_gr, subinit_gr, body_gr;
  schedule_ptr sched_list, sc;
  psdf_polynomial_list_ptr *set_par_list, set_par;
  int *val_ptr, val;
  psdf_polynomial_ptr par;
  user_values_ptr user_val;

  /* for each hierarchical actor, first schedule the init of that node
   */
  forall_nodes(nd, *gr) {
    nd_attr = gr->inf(nd);
    actor_attr_name_get(nd_attr.alist, actor_name);
    if (attr_exists(nd_attr.alist, actor_attr_psdf_init_graph)) {
      subgr = allocate_scalar(leda_actor_GRAPH_ptr);
```

```
      actor_attr_psdf_init_graph_get(nd_attr.alist, subgr);
      init_gr = *subgr;

      /* at this point delete from param_table all init-set and parent-set
       * parameter entries,
       * for this supernode. These entries correspond to an older run and are
       * now obsolete.
       */
      .....
      fprintf(fp, "computing schedule for init graph of subsystem %s\n", actor_name);
      psdf_run_sched_graph(fp, init_gr, user_table, param_table);
    }
    if (attr_exists(nd_attr.alist, actor_attr_psdf_is_subsystem))
      /* now compute interface token flow of this sub-system */
      psdf_run_compute_interface_token_flow(gr, nd, param_table);
  }
  /* now configure all the edges in this graph with integer values
   * based on the info in param_table, and compute the repetitions
   * vector
   */
  psdf_run_compute_reps(NULL, gr, param_table);
  /* construct valid schedule */
  sched_list = psdf_run_construct_valid_scehdule(gr);
  /* fire each actor from sched_list */
  for (sc = sched_list; sc; sc = sc->next) {
    act = sc->member;
    nd_attr = gr->inf(act);
    actor_attr_name_get(nd_attr.alist, actor_name);
    if (attr_exists(nd_attr.alist, actor_attr_psdf_is_subsystem)) {
      /* schedule subinit graph */
      if (attr_exists(nd_attr.alist, actor_attr_psdf_subinit_graph)) {
        subgr = allocate_scalar(leda_actor_GRAPH_ptr);
        actor_attr_psdf_subinit_graph_get(nd_attr.alist, subgr);
        subinit_gr = *subgr;
            fprintf(fp, "computing schedule for subinit graph of subsystem %s\n",
actor_name);
        psdf_run_sched_graph(fp, subinit_gr, user_table, param_table);
      }
      /* schedule body graph */
      if (attr_exists(nd_attr.alist, actor_attr_psdf_body_graph)) {
        subgr = allocate_scalar(leda_actor_GRAPH_ptr);
        actor_attr_psdf_body_graph_get(nd_attr.alist, subgr);
        body_gr = *subgr;
            fprintf(fp, "computing schedule for body graph of subsystem %s\n",
```

```
actor_name);
        psdf_run_sched_graph(fp, body_gr, user_table, param_table);
      }
      /* verify local synchrony */
      /* for each interface entry in interface list of this supernode,
       * compare the pre-computed token flow with the actual flow
       * and flag off error for any mismatch
       */
      psdf_run_check_interface_token_flow(gr, act, param_table);
      /* remove all internal parameters of this sub-system from param_table */
      .....
    } else {
      /* leaf actor, fire it */
      fprintf(fp, "fire actor %s\n", actor_name);
      /* check if it configures any parameter, if so, then assign a value to each
       * parameter set, and insert that into param_table. get the value
       * either from the user_table, or randomly
       */
      if (attr_exists(nd_attr.alist, actor_attr_psdf_sets_pars)) {
        set_par_list = allocate_scalar(psdf_polynomial_list_ptr);
        actor_attr_psdf_sets_pars_get(nd_attr.alist, set_par_list);
        for (set_par = *set_par_list; set_par; set_par = set_par->next) {
          par = psdf_polynomial_product(set_par->member,
                                psdf_polynomial_allocate_from_int(1));
          fprintf(fp, "Parameter ");
          psdf_polynomial_print(fp, par);
          /* read this parameter value from user table */
          if (user_val = (user_values_ptr)hashlst_find_element(user_table, (void *)par))
{
            /* extract user provided value from the table in variable val */
            .....
            fprintf(fp, "set to user value %d\n", val);
          } else {
            /* user hasn't provided a value, set to random value */
            val = psdf_set_param_random();
            fprintf(fp, "set to random value %d\n", val);
          }
          /* now insert this parameter, and its value into the param_table */
          .....
        }
      }
    }
  }
  return;
```

```
}

void psdf_run_compute_reps(attr_actor_ptr su_attr,
                           leda_actor_GRAPH_ptr gr,
                           hashlst_ptr param_table)
{
 /* this routine first configures every edge in the graph as an SDF edge
  * by resolving all parameter references in the token production and
  * consumption quantities, either from the param_table, else from default
  * values. Supernodes already have their token flow resolved.
  */

 psdf_polynomial_ptr *tf_ptr, pr, cn, dl;
 attr_arc ed_attr;
 leda_edge ed;
 int prd, cns, delay;
 attr_actor src_attr, snk_attr;
 leda_node src, snk;

 forall_edges(ed, *gr) {
  ed_attr = gr->inf(ed);
  tf_ptr = allocate_scalar(psdf_polynomial_ptr);
  arc_attr_psdf_tokens_produced_get(ed_attr.alist, tf_ptr); pr = *tf_ptr;
  arc_attr_psdf_tokens_consumed_get(ed_attr.alist, tf_ptr); cn = *tf_ptr;
  arc_attr_psdf_delay_get(ed_attr.alist, tf_ptr); dl = *tf_ptr;
  src = source(ed); snk = target(ed);
  src_attr = gr->inf(src);
  snk_attr = gr->inf(snk);
  /* if the source is a supernode, then the produced field is
   * already set in psdf_run_compute_interface_token_flow
   */
  if (!attr_exists(src_attr.alist, actor_attr_psdf_is_subsystem)) {
    prd = psdf_run_resolve_token_flow(su_attr, pr, param_table);
    arc_attr_sdf_tokens_produced_set(ed_attr.alist, prd);
  }
  if (!attr_exists(snk_attr.alist, actor_attr_psdf_is_subsystem)) {
    cns = psdf_run_resolve_token_flow(su_attr, cn, param_table);
    arc_attr_sdf_tokens_consumed_set(ed_attr.alist, cns);
  }
  /* now process the delay */
  delay = psdf_run_resolve_token_flow(su_attr, dl, param_table);
  arc_attr_sdf_delay_set(ed_attr.alist, delay);
  gr->assign(ed, ed_attr);
 }
```

```
  /* now all the edges have been configured as SDF edges, compute repetitions
   * vector
   */
  psdf_run_compute_sdf_reps(gr);
  return;
}

void psdf_run_compute_sdf_reps(leda_actor_GRAPH_ptr gr)
{
  /* this is a straightforward implementation of the procedure
   * ComputeRepetitions as given in "Software Synthesis of Dataflow Graphs",
   * by S. Bhattacharyya, et al., pg. 48.
   */
  ....
}

schedule_ptr psdf_run_construct_valid_scehdule(leda_actor_GRAPH_ptr gr)
{
  /* this is a straightforward implementation of the procedure
   * ConstructValidSchedule as given in "Software Synthesis of Dataflow Graphs",
   * by S. Bhattacharyya, et al., pg. 50.
   */
  ......
}
/**************** end psdf_run_time_schedule.c ****************/
```

GLOSSARY

$\perp$: The unspecified parameter value.

$Aparams(G)$: The subset of the parameters ($params(G)$) of a PSDF graph G, comprising all the unspecified actor parameters in G.

$Aparams(\Phi)$: The subset of the parameters ($params(\Phi)$) of a PSDF specification Φ, comprising all the unspecified actor parameters in Φ.

$actor(p)$: The PSDF actor to which port p belongs.

body condition:
A part of the requirement for local synchrony of a PSDF specification Φ — the body graph is inherently locally synchronous, and the token transfer at the interface ports of the body graph is invariant over the body graph parameters that are not configured in the init graph.

body graph: Same as Φ_b.

body inputs: Same as $inputs_b(\Phi)$.

body of S: Same as $body_S$.

$body_S$: The third part of a parameterized looped schedule S of a PSDSF graph G, called the *body of S*. It is a sequence $V_1 V_2 \ldots V_k$, where each V_i is either an actor (leaf or hierarchical) in G or a parameterized schedule loop.

bounded memory consistency:
A part of the requirement for local synchrony of a PSDF graph G — every possible instantiated SDF graph of G should satisfy the upper bounds specified by the max token transfer function at each actor port, and the max delay value on each edge.

$C(p)$: The value of parameter $p \in P$ under the configuration C of the non-empty parameter set P.

$c(e)$: The number of tokens consumed from the PSDF edge e, which is equivalent to the token consumption function of the actor to which the sink port of e belongs, corresponding to a complete configura-

tion of the parameters of e.

child specification of G:
Same as child subsystem of G.

child subsystem of G:
A PSDF specification $subsystem(H)$, associated with a hierarchical actor H in a PSDF graph G.

complete configuration:
A configuration C of a nonempty parameter set P such that for each $p \in P$, $C(p) \neq \perp$.

$config_A$: A configuration of $params(A)$, may be incomplete.

$config_{A, \bar{p}}$: The *instantiated configuration of* A *in* G *associated with the complete configuration* $\bar{p}$ of a PSDF graph G, which is obtained by "applying" the configuration $\bar{p}$ to unspecified parameters of A.

$config_e$: A configuration of $params(e)$, may be incomplete.

$config_{e, \bar{p}}$: The *instantiated configuration of* e *in* G *associated with the complete configuration* $\bar{p}$ of a PSDF graph G, which is obtained by "applying" the configuration $\bar{p}$ to unspecified parameters of e.

configuration: An assignment of values to parameters of a nonempty parameter set P, denoted by $\{(p_1, v_1), (p_2, v_2), \ldots, (p_n, v_n)\}$, such that for $i = 1, 2, \ldots, n$, $v_i \in (domain(p_i) \cup \{\perp\})$. An empty parameter set has an empty configuration.

configuration of G:
A configuration of the parameters ($params(G)$) of a PSDF graph G.

configuration of Φ:
A configuration of the parameters ($params(\Phi)$) of a PSDF specification Φ.

control hierarchy:
The hierarchy represented by a PSDF hierarchical actor that represents a PSDF specification.

$d(e)$: The number of delay tokens on the PSDF edge e, which is equivalent to δ_e,corresponding to a complete configuration of the parameters of e.

dataflow consistency:
A part of the requirement for local synchrony of a PSDF graph G — every possible instantiated SDF graph of G should have a valid schedule.

$default(p)$: The default value of parameter p, such that $default(p) \in (domain(p) \cup \{\perp\})$.

$domain(A)$: The domain of the parameter set ($params(A)$) of a PSDF actor A.

$\overline{domain(A)}$: The subset of $domain(A)$, comprising all the complete configurations of a PSDF actor A.

$domain(G)$: The domain of the parameter set of a PSDF graph G, defined as $Jdomain(params(G))$.

$\overline{domain(G)}$: The subset of $domain(G)$, comprising all the complete configurations of a PSDF graph G.

$domain(P)$: The set of compatible *valid* configurations of a nonempty parameter set P, in the context in which P is being used. An empty parameter set has an empty domain.

$\overline{domain(P)}$: The subset of $domain(P)$ of a parameter set P, that consists of all configurations that are both valid and complete.

$domain(e)$: The domain of the parameter set ($params(e)$) of a PSDF edge e.

$\overline{domain(e)}$: The subset of $domain(e)$, comprising all the complete configurations of a PSDF edge e.

$domain(p)$: A finite, nonempty set denoting the domain of a parameter p, that provides an enumeration of all the values that p can take on.

$domain(\Phi)$: The domain of the parameter set of a PSDF specification Φ, defined as the joint domain $Jdomain(params(\Phi))$

$\overline{domain(\Phi)}$: The subset of $domain(\Phi)$, comprising all the complete configurations of a PSDF specification Φ.

$Eparams(G)$: The subset of the parameters ($params(G)$) of a PSDF graph G, comprising all the unspecified edge parameters in G.

$Eparams(\Phi)$: The subset of the parameters ($params(\Phi)$) of a PSDF specification

Φ, comprising all the unspecified edge parameters in Φ

external subsystem parameter:
A subsystem parameters that control the behavior of the subsystem in such a way that they are visible externally in the parent graph of the subsystem. These parameters are configured in the associated init graph, or in actors of the parent graph of the subsystem.

external-dataflow parameter:
A parameter of the associated PSDF object (actor or subsystem) that affects the dataflow behavior of the object (number of tokens produced or consumed at actor ports or at subsystem interface ports).

external-nondataflow parameter:
A parameter of the associated PSDF object (actor or subsystem) that does not affect the dataflow behavior of the object (number of tokens produced or consumed at actor ports or at subsystem interface ports).

f_A: The *parameter interpretation function* of a PSDF actor A that implements κ_A and φ_A.

I_G: The *internally-connected input ports* of a PSDF graph G, comprising the set of input ports of actors that belong to G on which edges of G originate.

$IN(V_G)$: The set of actor input ports in a PSDF graph $G = (V_G, E_G)$.

immediate parameter of a subsystem:
Same as *subsystem parameter*.

$in(A)$: The set of input ports of a PSDF actor A.

inherently locally non-synchronous:
See *local synchrony of* G, and *local synchrony of* Φ.

inherently locally synchronous:
See *local synchrony of* G, and *local synchrony of* Φ.

inherited parameter of a subsystem:
Every subsystem inherits the immediate and inherited parameters of the parent subsystem of its parent graph, as its inherited parameters. Thus, the inherited parameters of a subsystem comprise the immediate parameters of all the ancestor subsystems.

init condition: A part of the requirement for local synchrony of PSDF specification Φ — the init graph is inherently locally synchronous, and it produces exactly one token at each output port on each invocation for every possible configuration of the init graph.

init graph: Same as Φ_i.

init-configured parameter:
A subsystem parameter that is configured in the associated init graph.

initChild phase of S:
The first part of a parameterized looped schedule S of a PSDSF graph G that consists of successive invocations of the parameterized looped schedules of the init graphs of the child subsystems of G.

initflow: The mechanism of configuring the parameters of a PSDF specification Φ ($params(\Phi)$) in the init and subinit graphs of hierarchically higher-level subsystems. This is distinct from dataflow.

initflow parameters:
Same as $params_{\Phi_S}(\Phi)$.

$inputs(G)$: The *interface inputs* of a PSDF graph G comprising the set of input ports of actors that belong to G at which no edges of G terminate.

$inputs(\Phi)$: The *interface inputs* of PSDF specification Φ, comprising all the dataflow inputs that go into Φ when Φ is embedded in a PSDF graph.

$inputs_b(\Phi)$: The *body inputs* of a PSDF specification Φ, comprising the set of dataflow inputs to Φ that appear as dataflow inputs to Φ_b.

$inputs_p(\Phi)$: The *parameter inputs* of a PSDF specification Φ, comprising the set of dataflow inputs to Φ that are bound to parameters of Φ_s.

$inputs_s(\Phi)$: The *subinit inputs* of a PSDF specification Φ, comprising the set of dataflow inputs to Φ that appear as dataflow inputs to Φ_s.

$instance_A(C)$:
A (pure) SDF actor obtained from a complete configuration C of a PSDF actor A.

$instance_G(\bar{p})$: The *instance of* G *associated with the complete configuration* $\bar{p}$. It denotes the SDF graph that emerges by "applying" the complete con-

figuration $\bar{p} \in \overline{domain(G)}$ to unspecified actor and edge parameters in the PSDF graph G.

internal subsystem parameter:
A subsystem parameters that controls the internal behavior of a subsystem, and is not visible externally in the parent graph of the subsystem. It is configured in the associated subinit graph.

internally-connected input ports:
Same as I_G.

internally-connected output ports:
Same as O_G.

invariant: For a parameter set P, a function $f: \overline{domain(P)} \rightarrow R$ into some range set R; and a subset $P' \subseteq P$, *f is invariant over P'* if the function f does not depend on any member of P'.

iteration count:Same as $loopcnt(L)$.

$Jdomain(\bar{P})$: The joint domain of $\bar{P}$, where $\bar{P}$ is a family of m disjoint, parameter sets $P_1, P_2, \ldots, P_m$, and is defined as the union of all valid configurations of each parameter set.

local synchrony of G:
A PSDF graph G is *inherently locally synchronous* if for every $\bar{p} \in \overline{domain(G)}$, the instantiated SDF graph $instance_G(\bar{p})$ has the following properties: it has a valid schedule (i.e. it is dataflow consistent and is deadlock free); it satisfies the upper bounds provided by the max token transfer function at each actor port, and the max delay value on each edge; and every child subsystem is locally synchronous. If no $\bar{p} \in \overline{domain(G)}$ satisfies these properties, then G is *inherently locally non-synchronous*. Otherwise, G is *partially locally synchronous*.

local synchrony of Φ:
A PSDF specification Φ is *inherently locally synchronous* (or *simply locally synchronous*), if each of Φ_i, Φ_s, and Φ_b is inherently locally synchronous; for each $\bar{p} \in \overline{domain(\Phi_i)}$, Φ_i produces one token on each output port on each invocation; for each $\bar{p} \in \overline{domain(\Phi_s)}$, Φ_s produces one token on each output port on each invocation; the token transfer on the interface input ports of Φ_s is invariant over the parameters of Φ_s that are bound to dataflow inputs of Φ; the token transfer on the interface ports of Φ_b is invariant over the parameters

of Φ_b that are not configured in Φ_i. If either of the graphs Φ_i, Φ_s, and Φ_b is locally non-synchronous, or the init and subinit graphs do not produce one token at each output port for any valid configuration of the respective graph parameters, then Φ is *inherently locally non-synchronous* (or *simply locally non-synchronous*). Otherwise, Φ is *partially locally synchronous*.

loopcnt(L): The iteration count of a parameterized schedule loop L in PSDF graph G, denoting the number of repetitions of the invocation sequence $T_1 T_2 \ldots T_m$ associated with L. In general, it is a symbolic expression consisting of constants, compiler-generated variables, and subsystem parameters.

max token transfer function:
See τ_A.

max delay value:
See μ_e.

non-init-configured parameter:
A subsystem parameter that is not configured in the associated init graph. The set of non-init-configured parameters comprise the internal subsystem parameters and parent-configured subsystem parameters.

O_G: The *internally connected output ports* of a PSDF graph G comprising the set of output ports of actors that belong to G at which edges of G terminate.

$OUT(V_G)$: The set of actor output ports in a PSDF graph $G = (V_G, E_G)$.

out(A): The set of output ports of a PSDF actor A.

outputs(G): The *interface outputs* of a a PSDF graph G comprising the set of output ports of actors that belong to G at which no edges of G originate.

outputs(Φ): The *interface outputs* of PSDF specification Φ, comprising all the dataflow outputs that go out of Φ when Φ is embedded in a PSDF graph. These outputs of Φ are simply the outputs of the associated body graph Φ_b.

P-APGAN: The clustering technique used for quasi-static scheduling of acyclic PSDF graphs, called *parameterized APGAN* (Acyclic Pairwise

Grouping of Adjacent Nodes).

P-APGAN candidate:
Two adjacent actors in a PSDF graph, clustering of which through P-APGAN does not introduce a cycle in the graph.

PSDF actor, A:
A PSDF actor with a set of input and output ports, and a parameter set.

PSDF edge, e:A PSDF edge with a parameter set, connecting a PSDF actor output port to a PSDF actor input port.

PSDF graph G:
A bipartite, directed graph, represented by an ordered pair (V_G, E_G), where V_G is a set of PSDF actors, and E_G is a set of PSDF edges that connect a subset of the actor output ports to a subset of the set of input ports.

PSDF hierarchical actor:
A PSDF actor that represents a PSDF specification Φ.

PSDF specification, Φ:
A PSDF specification or subsystem, composed of three PSDF graphs — the init graph, the subinit graph, and the body graph.

PSDF subsystem:
Same as PSDF specification.

$p(e)$: The number of tokens produced onto the PSDF edge e, which is equivalent to the token production function of the actor to which the source port of e belongs, corresponding to a complete configuration of the parameters of e.

parameter inputs:
Same as $inputs_p(\Phi)$.

parameter set P:
A finite set of objects $\{p_1, p_2, \ldots, p_n\}$, where each p_i is called a *parameter* of P.

parameterized looped schedule S_G:
A schedule for a PSDF graph G, consisting of three parts — the *init-child phase of* S, the *preamble of* S, and the *body of* S.

parameterized schedule loop L:
For a PSDF graph G, it represents successive repetition of an invocation sequence $T_1 T_2 \ldots T_m$, where each T_i is either a leaf actor in G, or a hierarchical actor in G, or another parameterized schedule loop.

$params(A)$: The parameter set of a PSDF actor A.

$params(G)$: The *parameter set* of a PSDF graph G, which comprises all the unspecified actor parameters and edge parameters in G.

$params(e)$: The parameter set of a PSDF edge e.

$params(\Phi)$: The *parameter set* of a PSDF specification Φ, called the *specification parameters* of Φ. It comprises the initflow parameters of Φ_s, and the parameters of Φ_i. These are set in the init and subinit graphs of hierarchically higher level subsystems.

$params_A(G)$: The *set of unspecified parameters of* a PSDF actor A in a PSDF graph G, obtained as the set $\{(A, p)\}$ such that $(config_A(p) = \perp)$, if A has a nonempty parameter set, and otherwise it is defined to be empty.

$params_e(G)$: The *set of unspecified parameters of* a PSDF edge e in a PSDF graph G, obtained as the set $\{(e, p)\}$ such that $(config_e(p) = \perp)$, if e has a nonempty parameter set, and otherwise it is defined to be empty.

$params_{\Phi_S}(\Phi)$:
The *initflow* parameters of Φ_s comprising those parameters of Φ_s that are not bound to an interface output port of Φ_i, nor to a dataflow input of Φ.

parent graph of Φ:
The PSDF graph to which hierarchical actor H belongs, where $\Phi = subsystem(H)$.

parent specification of G:
The PSDF specification with which the PSDF graph G is associated as an init graph, subinit graph, or body graph.

parent subsystem of G:
Same as parent specification of G.

parent-configured parameter:

A subsystem parameter that is configured in an actor of the parent graph of the subsystem.

partially locally synchronous:
See *local synchrony of* G, and *local synchrony of* Φ.

preamble of S: Same as $preamble_S$.

$preamble_S$: The second part of a parameterized looped schedule S of a PSDSF graph G, called the *preamble of* S. It consists of code that configures the iteration count of each schedule loop in $body_S$ by defining in a proper order every compiler-generated variable used in the symbolic expression of the iteration count, and includes conditionals for checking sample rate consistency, and bounded memory consistency of graph G. Additionally, if G is an init graph, $preamble_S$ includes conditionals for checking the init condition for local synchrony of the parent specification of G. Similarly, if G is a subinit graph, then $preamble_S$ includes conditionals for checking the subinit output condition for local synchrony of the parent specification.

projection: For a nonempty parameter set P, a configuration C of P, and a nonempty subset of parameters $P' \subseteq P$, the *projection of* C *onto* P' is obtained by discarding from C all elements containing parameters that do not belong to P'. Projecting a configuration (empty or nonempty) onto an empty configuration produces the empty configuration.

$\mathbf{q}_{G,\bar{p}}$: The *parameterized repetitions vector* associated with the instantiated SDF graph $instance_G(\bar{p})$, for a complete configuration $\bar{p} \in domain(G)$ of a PSDF graph G.

quasi-static schedule:
A schedule that is computed at compile-time, but contains code for performing some data-dependent computations at run-time.

refinement: For a nonempty parameter set P, a nonempty subset of parameters $P' \subseteq P$, a configuration of P, $C \in domain(P)$, and a configuration of P', $C' \in domain(P')$, the *refinement of* C *with respect to* C' is obtained by augmenting the configuration C by assigning the value $C'(p)$ to all parameters of P' that are unspecified in C, but specified in C'. Refining a configuration (empty or nonempty) with respect to an empty configuration, maintains the original configuration.

semantic hierarchy:
Same as *control hierarchy*.

simple cyclic graph:
A cyclic PSDF graph, where each fundamental directed cycle has a single delay element with a known value, and each such fundamental cycle is statically known to be a single-rate system (i.e. $p(e) = c(e)$ for each edge e in the fundamental cycle), or can be configured into a single-rate system.

simplified PSDF graph:
A directed multigraph (V_G, E') associated with the PSDF graph $G = (V_G, E_G)$, where $E' = \{(actor(src(e)), actor(snk(e)))|(e \in E_G)\}$. In other words, edges are interpreted to connect actors, instead of connecting actor ports.

$snk(e)$: The PSDF actor input port at which PSDF edge e terminates. Overloaded in the simplified PSDF graph to indicate the actor at which PSDF edge e terminates.

$src(e)$: The PSDF actor output port at which PSDF edge e originates. Overloaded in the simplified PSDF graph to indicate the actor at which PSDF edge e originates.

subinit graph: Same as Φ_s.

subinit input condition:
A part of the requirement for local synchrony of a PSDF specification Φ — the subinit graph is inherently locally synchronous, and the token transfer at the interface input ports of the subinit graph is invariant over the subinit graph parameters that are not set in the init graph.

subinit inputs: Same as $inputs_s(\Phi)$.

subinit output condition:
A part of the requirement for local synchrony of a PSDF specification Φ — the subinit graph is inherently locally synchronous, and it produces exactly one token at each output port on each invocation for every possible configuration of the subinit graph parameters.

subinit-configured parameter:
Same as *internal subsystem parameter*.

subsystem parameter:
A parameters of a PSDF specification or subsystem, directly derived from the application, while modeling an application in PSDF. Actor and edge parameters in a PSDF graph are configured with subsystem parameters.

$subsystem(H)$:
The PSDF subsystem (specification) associated with a PSDF hierarchical actor H.

syntactic hierarchy:
The hierarchy represented by a PSDF actor (subblock), where the internals of the actor is represented by another PSDF graph. This hierarchy can be flattened and the subblock can be replaced with the graph that it represents.

$ToBody(\Phi_i)$: The set of interface output ports of Φ_i that are used to set parameters of the associated body graph Φ_b.

$ToSubinit(\Phi_i)$:
The set of interface output ports of Φ_i that are used to set parameters of the associated subinit graph Φ_s.

unit transfer consistency:
The input condition and subinit output condition for local synchrony of a PSDF specification Φ.

VLS_b: A block of code to verify the body condition for local synchrony of a PSDF specification Φ that appears as a part of a parameterized schedule loop in the parent graph of Φ.

VLS_s: A block of code to verify the subinit input condition for local synchrony of a PSDF specification Φ that appears as a part of a parameterized schedule loop in the parent graph of Φ.

γ_e: The *delay re-initialization period function* which specifies the re-initialization period of a delay token for any complete, valid configuration of the parameter set of a PSDF edge e.

δ_e: The *delay function* that gives the number of delay tokens on a PSDF edge e for a complete valid configuration of e.

κ_A: The *port consumption function* of a PSDF actor A, that is defined as $\kappa_A : (in(A) \times \overline{domain(A)}) \rightarrow Z^+$.

μ_e: The *max delay value* on a PSDF edge e specifying an upper bound for dynamically varying delay tokens on e.

τ_A: The *max token transfer function* of a PSDF actor A that specifies an upper bound on the maximum number of tokens transferred (produced or consumed) at a port of A.

υ_e: The *delay initial value function* of a PSDF edge e which gives the initial value associated with a delay token for a complete valid configuration of the parameters of e.

Φ_b: The body graph of a a PSDF specification Φ.

Φ_{fs}: The function that maps each output port of the subinit graph Φ_s to a parameter in the associated body graph Φ_b.

Φ_i: The init graph of a PSDF specification Φ.

Φ_{ps}: The function that maps each member of $inputs_p(\Phi)$ to a parameter of Φ_s.

Φ_s: The subinit graph of a PSDF specification Φ.

Φ_{tb}: The function that maps each member of $ToBody(\Phi_i)$ to a parameter in Φ_b.

Φ_{ts}: The function that maps each member of $ToSubinit(\Phi_i)$ to a parameter in Φ_s.

φ_A: The *port production function* of PSDF actor A, and is defined as $\varphi_A : (out(A) \times \overline{domain(A)}) \rightarrow Z^+$.

REFERENCES

[1] M. Ade, R. Lauwereins and J.A. Peperstraete, "Data Memory Minimization for Synchronous Data Flow Graphs Emulated on *DSP-FPGA* Targets," *Proceedings of the Design Automation Conference*, June, 1994.

[2] R. Allen, and D. Kennedy, "Automatic Transformations of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, **Vol. 9**, **No. 4**, October, 1987.

[3] S. Banerjee, T. Hamada, P. M. Chau, and R. D. Fellman, "Macro Pipelining Based Scheduling on High Performance Heterogeneous Multiprocessor Systems," *IEEE Transactions on Signal Processing*, **Vol. 43**, **No. 6**, June, 1995.

[4] S.S. Bhattacharyya, J.T. Buck, S. Ha, and E.A. Lee, "Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms," *IEEE Transactions on Circuits and Systems — I: Fundamental Theory and Applications*, March, 1995.

[5] S.S. Bhattacharyya and E.A. Lee, "Looped Schedules for Dataflow Descriptions of Multirate Signal Processing Algorithms," *Journal of Formal Methods in System Design*, December, 1994.

[6] S. S. Bhattacharyya, and E. A. Lee, "Memory Management for Dataflow Programming of Multirate Signal Processing Algorithms," *IEEE Transactions on Signal Processing*, **Vol. 42**, **No. 5**, May, 1994.

[7] S. S. Bhattacharyya, R. Leupers and 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.

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

[9] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Optimizing Synchronization in Multiprocessor DSP Systems," *IEEE Transactions on Signal Processing*, **Vol. 45**, **No. 6**, June, 1997.

[10] J.T. Buck, "Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model," Tech. Report UCB/ERL 93/69, Ph.D. thesis, University of California at Berkeley, September,1993.

[11] J.T. Buck, “Static Scheduling and Code Generation from Dynamic Dataflow Graphs With Integer-Valued Control Streams,” *28th Asilomar Conference on Signals, Systems, and Computers*, November,1994.

[12] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *International Journal of Computer Simulation*, **Vol. 4**, April, 1994.

[13] L. F. Chao, and E. Sha, “Unfolding and Retiming Data-Flow DSP Programs for RISC Multiprocessor Scheduling,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April, 1992.

[14] G.A. Clark, S.K. Mitra, and S.R. Parker, “Block Implementation of Adaptive Digital Filters,” *IEEE Transactions on Circuits Syst.*, **Vol. CAS-28**, 1981.

[15] M. Engels, G. Bilsen, R. Lauwreins, J. Peperstraete, “Cyclo-Static dataflow,” *IEEE Transactions on Signal Processing*, **Vol. 44**, **No. 2**, February 1996.

[16] L. Freund, M. Israel, F. Rousseau, J. M. Berge, M. Auguin, C. Belleudy, and G. Gogniat, “A Codesign Experiment in Acoustic Echo Cancellation: GMDFα,” *ACM Transactions on Design Automation of Electronic Systems*, **Vol. 2**, **No. 4**, October, 1997.

[17] G.R. Gao, R. Govindarajan, and P. Panangaden, “Well-Behaved Programs for DSP Computation,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, March, 1992.

[18] 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.

[19] D. Harel, “StateCharts: A Visual Formalism for Complex Systems,” *Sci. Comput. Program.*, **Vol. 8**, 1987.

[20] M. Hayes, *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, Inc., 1996.

[21] S. Haykin, *Adaptive Filter Theory*, 3rd edition, Prentice Hall Information and System Sciences Series, 1996.

[22] P. Hoang, and J. Rabaey, “A Compiler for Multiprocessor DSP Implementation,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, March, 1992.

[23] E.A. Lee, “Consistency in Dataflow Graphs,” *IEEE Transactions on Parallel and Distributed Systems*, **Vol. 2**, **No. 2**, April, 1991.

[24] E. A. Lee, “Representing and Exploiting Data Parallelism Using Multidimensional Dataflow Diagrams,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April, 1993.

[25] E.A. Lee, “Static Scheduling of Data-Flow Programs for DSP,” *Advanced Topics in Data-Flow Computing*, ed. J. Gaudiot, L. Bic, Prentice Hall, Engelwood Cliffs, NJ, 1991.

[26] E.A. Lee and D.G. Messerschmitt, “Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing,” *IEEE Transactions on Computers*, **Vol. C-36, No. 2**, February, 1987.

[27] K. Mehlhorn, S. Naher, M. Seel, C. Uhrig, “The LEDA User Manual,” Version 3.7, *http://www.mpi-sb.mpg.de/LEDA*.

[28] M. Pankert, O. Mauss, S. Ritz, H. Meyr, “Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April, 1994.

[29] K. K. Parhi, and D. G. Messerschmitt, “Static Rate-optimal Scheduling of Iterative Data-flow Programs via Optimum Unfolding,” *IEEE Transactions on Computers*, **Vol. 40**, **No. 2**, February, 1991.

[30] J.L Pino, S.S. Bhattacharyya, and E.A. Lee, “A Hierarchical Multiprocessor Scheduling System for DSP Applications,” *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, November, 1995.

[31] J. J. Shynk, “Frequency-Domain and Multirate Adaptive Filtering,” *IEEE Signal Processing Magazine*, **Vol. 9**, **No. 1**, January, 1992.

[32] W. Sung, J. Kim and S. Ha, “Memory Efficient Synthesis from Dataflow Graphs,” *Proceedings of the International Symposium on Systems Synthesis*, 1998.

[33] S. Ritz, M. Pankert and H. Meyr, “Optimum Vectorization of Scalable Synchronous Dataflow Graphs,” *Proceedings of the International Conference on Application Specific Array Processors*, October, 1993.

[34] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall Signal Processing Series, 1993.

[35] P. Wauters, M. Engels, R. Lauwereins, J.A. Peperstraete, “Cyclo-dynamic dataflow,” *4th EUROMICRO Workshop on Parallel and Distributed Processing*, January, 1996.

[36] E. Zitzler, J. Teich and S. S. Bhattacharyya, “Evolutionary Algorithms for the Synthesis of Embedded Software,” to appear in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1999.