EFFICIENT DESIGN OF EMBEDDED SIGNAL PROCESSING SYSTEMS USING TOPOLOGICAL PATTERNS BASED DATAFLOW GRAPH REPRESENTATIONS

by

Nimish Sane

Report submitted to the
Salzburg University of Applied Sciences, Salzburg, Austria,
in partial fulfillment of the requirements for the
Marshall Plan Scholarship
2011

Advisor:
Professor Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park, USA.

ABSTRACT

Tools for designing signal processing systems with their semantic foundation in dataflow modeling often use high-level graphical user interfaces (GUIs) or text based languages that allow specifying applications as directed graphs. Such graphical representations serve as an initial reference point for further analysis and optimizations that lead to platform-specific implementations. For large-scale applications, the underlying graphs often consist of smaller substructures that repeat multiple times. To enable more concise representation and direct analysis of such substructures in the context of high level digital signal processing (DSP) specification languages and design tools, we have developed the modeling concept of *topological patterns*, and proposed ways for supporting this concept in a high-level language. This report shows how the dataflow interchange format (DIF) language can be augmented with constructs for supporting topological patterns, and topological patterns can be effective in various aspects of embedded signal processing design flows using specific application examples.

Acknowledgments

I express my sincere gratitude toward my advisor Prof. Shuvra S. Bhattacharyya for his guidance and support throughout this research work. His advice, technical inputs, and feedback have been extremely valuable.

I thank the Austrian Marshall Plan Foundation for awarding me the Marshall Plan Scholarship (MPS). The MPS has supported the research presented in this report. It allowed me to participate in the semester-long student exchange opportunity at the Salzburg University of Applied Sciences/ Fachhochschule Salzburg (FHS), Salzburg, Austria. Personally, this was one of my most memorable experiences.

The research presented in this report was also sponsored in part by the US Air Force Research Laboratory (AFRL-FA87501110049), Laboratory for Telecommunication Sciences, National Radio Astronomy Observatory, and US National Science Foundation. I acknowledge them with thanks for their support.[1]

I am thankful to the FHS for hosting me as an MPS recipient. I thank Prof. DI Dr. Gerhard Jöchtl, Head, Degree Program in Information Technology and Systems Management (ITS) at the FHS for his personal attention in ensuring an excellent research environment, and access to facilities at the FHS. I also thank the other faculty and staff of the ITS program at the FHS. In particular, I am thankful to DI Simon Kranzer, DI Dr. Robert Merz, DI Sabine Klausner, Prof. Dr. Karl Entacher, and Prof. Dr. Stefan Wegenkittl for not only the wonderful technical discussions but also their warm hospitality,

[1]I acknowledge Government's support in publication of this report. Any opinions, findings, and conclusions or recommendations expressed in this report are those of the author and do not reflect the views of AFRL, LTS, NRAO, or NSF.

and support. I thank Sandra Lagler, and Sonja Treiber for their support. I also thank Andreas Unterweger, Peter Ott, Benjamin Lachmayer, Carlos Jambrina, and Bernadette Himmelbauer of the ITS program at the FHS for their wonderful company.

I am grateful to Prof. Mag. Dr. Gabriele Abermann for her excellent assistance in facilitating this student exchange at the FHS, and co-ordinating the MPS related arrangements. Without her and others from the International Office at the FHS, the transition into the environment at the FHS, and in Austria, in general, would not have been so smooth. I also thank Laura Streitbürger, Mag. Teresa Rieger, Rosalyn Eder, and Ines Aufschnaiter for the same.

I would also like to thank Dr. Hojin Kee (of National Instruments, Inc., Austin, Texas, USA), Dr. Gunasekaran Seetharaman (of the US Air Force Research Laboratory, Rome, New York, USA), and Shenpei Wu (of the University of Maryland, College Park, Maryland, USA) for their valuable contributions to some aspects of this research.

Table of Contents

Chapter 1

Introduction

Dataflow modeling is used extensively for designing signal processing systems. There are various existing design tools with their semantic foundations in dataflow modeling such as Agilent ADS [25], National Instruments LabVIEW [2], Compaan/Laura [31], and SysteMoc [13]. DSP-oriented dataflow design tools typically allow high-level application specification, software simulation, and possibly synthesis for hardware or software implementation. These tools employ high-level description languages for application specification. These languages, which may be either GUI or text based, provide syntactic and semantic constructs for specifying graphical representations of DSP applications. Such graphical representations are then parsed and converted into intermediate representations suitable for further processing.

In this work, we address the problem of representing large-scale and scalable dataflow graphs that have complex topologies. Such graphs comprise of various kinds of functional substructures that are parameterizable and can be represented in terms of concise, scalable specifications.

For example, the dataflow graph of an N-point fast Fourier transform (FFT) algorithm consists of a combination of scaled versions of a well-known pattern called the *butterfly diagram* [24], and a systolic array is a *mesh* of computing elements having a specific dataflow structure that can solve problems such as QR-decomposition based

recursive least square adaptive filtering, and minimum variance distortionless response beamforming [19]. We identify such common structures in dataflow graphs as *topological patterns,* and treat this kind of pattern as a first class citizen in the modeling process. Furthermore, we demonstrate and experiment with the use of topological patterns in the DIF, a textual design language and associated software package for specification, analysis, and synthesis based on DSP-oriented dataflow models of computation [15], [26].

Topological patterns not only permit scalable specifications of dataflow substructures but also expose the underlying graph structure explicitly to the corresponding design tool. This allows design tools to exploit any analysis or optimization advantages offered by the substructures without having to "discover" those structures through additional levels of pre-processing analysis. Some of the key components of the design flow that can potentially benefit from explicitly exposed patterns include various kinds of scheduling transformations, and techniques for buffer memory optimization. Furthermore, by making it easier and more efficient to apply substructure-specific analysis techniques, programming support for topological patterns encourages the development of such analysis techniques, and provides a natural interface for reusing them across different applications and tools.

In this report, we provide background on dataflow modeling, and the DIF language as well as discuss the relevant prior work in Chapter 2. The concept of topological patterns is elaborated in Chapter 3, including a description of how we extend the DIF language to integrate topological patterns as a first class modeling construct. In Chapter 4, we show how topological patterns can be used by dataflow based design tools for dataflow graph analysis and transformations. We show how topological patterns can be used for

graph analysis; extracting implementation-specific features; representing schedules; and experimenting with pattern-specific schedules.

Chapter 2

Background and Related Work

This chapter provides background on dataflow modeling and the DIF language. We also discuss earlier research efforts that are relevant to this work.

2.1 Dataflow Modeling

Dataflow modeling involves representing an application using a directed graph $G = (V, E)$, where V is a set of vertices (nodes) and E is a set of edges. Each vertex $u \in V$ in a dataflow graph is called an *actor*, and represents a specific computation block, while each directed edge $(u, v) \in E$ is a first-in-first-out (FIFO) buffer that represents a communication link between the *source* actor u and the *sink* actor v. A dataflow graph edge e can also have a non-negative integer *delay*, $\mathrm{del}(e)$, associated with it, which represents the number of initial data values (*tokens*) present in the associated buffer.

Dataflow graphs operate based on *data-driven execution*, where an actor can be executed (*fired*) whenever it has sufficient amounts of data (numbers of "samples" or "data tokens") available on all of its inputs. During each firing, an actor consumes a certain number of tokens from each input and produces a certain number of tokens on each output.

In synchronous dataflow (SDF), these numbers are constant across all actor firings for a given input or output [21]. In SDF graphs, we refer to these numbers of tokens

consumed and produced in each actor execution as the *consumption rate* and *production rate* of the associated input and output, respectively. SDF is an especially popular form of dataflow that is used in many DSP-oriented design tools.

For a dataflow graph edge e, $\mathrm{src}(e)$ and $\mathrm{snk}(e)$ denote the source actor and sink actor of the edge, respectively. Additionally, if e is an SDF edge, then $\mathrm{prd}(e)$ represents the number of tokens produced on the edge by each firing of $\mathrm{src}(e)$, while $\mathrm{cns}(e)$ represents the number of tokens consumed from the edge by each firing of $\mathrm{snk}(e)$.

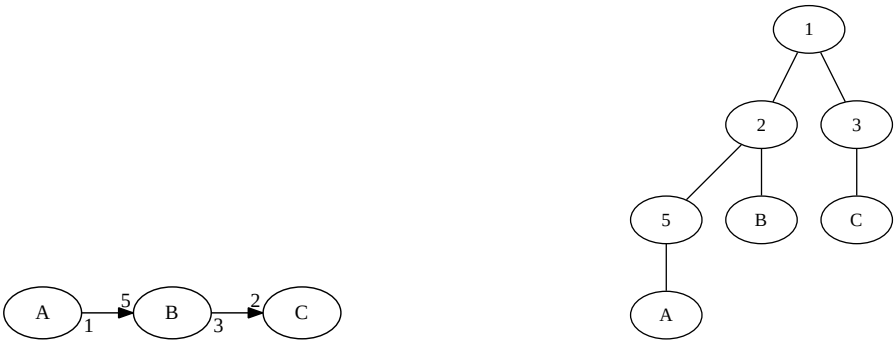
Usually production and consumption rate information is characterized in terms of individual input and output ports so that each port of an actor can in general have a different production or consumption rate characterization. Such characterizations can have constant values as in SDF [21]; periodic patterns of constant values as in cyclo-static dataflow (CSDF) [6]; or more complex forms that are data-dependent (e.g., see [7], [4], and [26]).

A *schedule* for a dataflow graph G is a sequence of actors in G, and represents the order in which actors are fired during an execution of G. In case of SDF graphs, it is possible to construct a periodic schedule that repeats itself during application execution. In the rest of the report, by a "schedule" for an SDF graph, we mean a periodic schedule. Each actor $u \in V$ fires exactly $q(u)$ times in a periodic schedule, where $q(u)$ is its repetition count which is obtained by solving the balance equation

$$q(\mathrm{src}(e)) \times \mathrm{prd}(e) = q(\mathrm{snk}(e)) \times \mathrm{cns}(e) \tag{2.1}$$

for each edge $e \in E$ [21].

For example, consider the SDF graph shown in Fig. 2.1(a). The repetition counts



(a) (b)

Figure 2.1: (a) An SDF graph for a sample rate converter. (b) A schedule for this graph that is represented using a GST.

for actors A, B, and C in this graph are 10, 2, and 3, respectively. A flat schedule for an SDF graph consists of firing every actor as many times as its repetition count in an order given by the topological sort of the application graph. A flat schedule for the SDF graph in Fig. 2.1(a) is given by $(10\ A)(2\ B)(3\ C)$, where $(n\ X)$ specifies n successive invocations of a schedule element (possibly an actor) X. It is possible to construct a schedule having nested loops that generally has fewer number of tokens accumulated on buffer edges during its execution. One nested looped schedule for the SDF graph in Fig. 2.1(a) is given by $(2\ (5\ A)B)(3\ C)$. In both of these schedules, every actor appears only once. We refer to such schedules as single appearance schedules.

2.2 Generalized Schedule Trees

A schedule for an application dataflow graph obtained after analyzing the graph is often represented using a graphical structure called a generalized schedule tree (GST). GSTs provide a dataflow-model-independent representation of schedules, which can be

utilized as an input to subsequent stages of the design flow, such as simulation and code synthesis [18]. GSTs are ordered trees with leaf nodes pointing to the actors of the associated application dataflow graph. An internal node of a GST denotes a loop count (an iteration construct to be applied when executing the schedule). We denote the loop count and actor associated with a node u in a GST by $\mathrm{count}(u)$ and $\mathrm{actor}(u)$, respectively. The GST representation allows exploiting topological information and algorithms for ordered trees in order to access and manipulate schedule elements. The execution of a schedule involves traversing the GST in a depth-first manner, and during this traversal, the subschedule rooted at any internal node is executed as many times as specified by the loop count of that node. Fig. 2.1(b) shows a GST for a valid looped schedule for the SDF graph shown in Fig. 2.1(a). This particular GST represents the firing sequence $(2\ (5\ A)B)(3\ C)$.

2.3 The Dataflow Interchange Format

To describe dataflow applications for a wide range of DSP applications, application developers can use the DIF language, which is a standard language founded in dataflow semantics and tailored for DSP system design [15]. DIF provides an integrated set of syntactic and semantic features that can fully capture essential modeling information of DSP applications without over-specification. From a dataflow point of view, DIF is designed to describe mixed-grain graph topologies and hierarchies as well as to specify dataflow-related and actor-specific information. The dataflow semantic specification is based on dataflow modeling theory and independent of any design tool.

Fig. 2.2 illustrates some of the available constructs in the DIF language along with

```
[dataflowModel] graphID {
    basedon {
        graphID;
    }

    [topology] {
        nodes = nodeID, ...;
        edges = edgeID(srcNodeID, snkNodeID), ...;
    }

    [builtInAttribute] {
        elementID = value;
        elementID = id;
        elementID = id1, id2, ...;
    }

    [attribute] userDefinedAttribute {
        elementID = value;
        elementID = id;
        elementID = id1, id2, ...;
    }
}
```

Figure 2.2: The DIF language

the syntax used for application specification. More details on the DIF language can be found in [15]. The `topology` block of a DIF specification specifies the graph topology, which includes all of the nodes and edges in the graph. DIF supports built-in attributes such as annotations that give the production and consumption rate constants for SDF edges. These pre-defined attributes are designated through special keywords in the language. DIF also allows user-defined attributes, which have a similar syntax as built-in attributes except that they need to be declared with the `attribute` keyword.

To facilitate use of the DIF language, the DIF package (TDP) has been built. Along with the ability to transform DIF descriptions into manipulable internal representations, TDP contains graph utilities, optimization engines, verification techniques, a comprehen-

sive functional simulation framework, and a software synthesis framework for generating C code [15], [26]. These facilities make TDP an effective environment for modeling dataflow applications, providing interoperability with other design environments, and developing and experimenting with new tools and dataflow techniques. Beyond these features, DIF is also suitable as a design environment for implementing dataflow-based application representations. Describing an application graph is done by listing nodes and edges, and then annotating dataflow specific information.

2.4 Related Work

Block diagrams are a natural and convenient way of describing DSP algorithms, and hence, DSP systems designers find it intuitive to have a high-level application specification that captures such a description. GUI based dataflow languages try to capture this intuition using visually appealing representations, while text based languages provide syntax that looks similar to common procedural languages, such as C, but with semantic constructs that model the dataflow structure of DSP block diagrams. To effectively handle the increasing complexity of signal processing system design, these languages must provide frameworks for modular and scalable representations with sufficient expressive power.

Earlier research efforts have focused on supporting commonly used and highly expressive constructs from procedural languages, such as recurrences, iteration, and conditionals, in dataflow-oriented languages [20]. Subsequent work includes evolution of various textual languages for DSP system design, such as SILAGE [33], StreamIt [32], and

CAL [11]. The StreamIt language provides high-level, architecture-independent abstractions for streaming applications geared toward large-scale program development. The CAL language is an actor-oriented language, which has been applied actively for field programmable gate array (FPGA) implementation and reconfigurable video coding applications. The SILAGE language has been developed with an emphasis on support for high level synthesis and multidimensional signal processing.

While these previous efforts have employed useful techniques for deriving and exploiting various types of specialized dataflow substructures within their respective compilers, they lack a general method for explicit and scalable representation of such substructures by the programmer. Such a programming interface for topological patterns is essential to capture the broad range of relevant patterns in ways that are scalable, and flexibly extensible to accommodate new types of patterns as they emerge from new applications and modeling techniques. Our concept of topological patterns is designed precisely to bridge this gap.

In other prior work, higher-order functions have been shown to permit elegant construction of structured subsystems in dataflow representations [23]. Higher-order functions are functions that take functions as inputs or produce functions as outputs. Topological patterns provide a related but technically different approach since topological patterns operate on generic directed graph vertices (e.g., `nodes` in DIF), where the actual binding to actor functionality and associated actor parameter values is specified separately, possibly through additional *parameter propagation patterns* (*PPP*s) [28]. Thus, unlike higher-order functions that take functions as arguments, topological patterns take only generic graph vertices (or arrays of such vertices) as arguments. Furthermore, our devel-

opment of topological patterns is tightly integrated with textual graph representation and arrays of graph vertices and edges, which are useful for providing scalable representations and managing large-scale designs.

Perhaps the most closely related prior work is that on support for arrays of vertices and edges in the DIF language with array construction syntax and semantics similar to those in the C language [9]. These constructs provide a useful shorthand notation for specifying related groups of graph elements (nodes or edges) as arrays in which individual elements can be easily indexed. A typical `elementID` in the DIF specification (see Fig. 2.2) when referred to as `baseName[N]`, generates an array of N elements. For example, `tap[N]` in DIF specifies an array `tap` of N nodes. The ith node, where $i = 0, 1, \ldots, N-1$, can be accessed using its index as `tap[i]`. However, in this *first-version* array support within DIF, there is no mechanism for instantiating (declaring) collections of related edges automatically as structured mappings among corresponding subsets of nodes. It is also not possible to configure parameters across arrays of actors as functions of the array indices. These two features — scalable, programmatic instantiation of graphical substructures, and association of parameter values — are provided by our development of topological patterns.

This development is orthogonal to the existing support for syntactic and semantic hierarchy in the DIF language, which allows constructing hierarchical dataflow graphs. The focus here is to allow the designer to specify already identified topological patterns in the design and expose such patterns to the enclosing design tool or design process, which is generally not achieved through conventional methods for using hierarchical dataflow graphs.

This report presents formulation of the concept of topological patterns and its application to dataflow modeling. To prototype this concept in DIF, we build upon the first-version framework of arrays in DIF, and introduce new modeling and language constructs that are dedicated to topological patterns. We also demonstrate the use of topological patterns to derive efficient implementations.

An initial formulation of topological patterns was presented in [28], where applications of topological patterns to representing equivalent homogeneous SDF (HSDF) graphs of SDF and CSDF application graphs was also presented, as well as trade-off analysis for an FPGA implementation of a JPEG image compression application. The work presented here goes beyond the developments of [28] by significantly expanding the exploration of application scenarios for topological patterns. Specifically, we explore the utility of topological patterns in analyzing dataflow graphs and extracting implementation-specific features. We also use topological patterns to represent schedules obtained after applying scheduling transformations to dataflow graphs, and derive more efficient implementations from such representations. Additionally, we show how specific topological patterns can be exploited to construct structured schedules, and how designers can experiment with corresponding scheduling trade-offs. A version of this work was published recently in [29].

Chapter 3

Topological Patterns

We have developed the concept of topological patterns for concise specification of functional structures at the dataflow graph (inter-actor) level. Topological patterns provide a scalable approach to specifying regular functional structures in a manner that is analogous in some ways to the use of design patterns in object oriented software [12], but with additional properties associated with being formally integrated with the framework of dataflow. This integration allows not only for specification of functional patterns but also for their analysis and optimization as part of the larger framework of dataflow.

Topological patterns build on the concepts of *graph element arrays*, which allow indexed families of graph elements to be declared and treated as single units for purposes of graph construction and analysis. As with arrays in conventional programming languages, graph element arrays can be single- or multi-dimensional. Additionally, they can be parameterized in terms of dataflow graph attributes so that their sizes and other characteristics can be conveniently adapted.

3.1 Topological Patterns in Signal Processing

We motivate the utility of incorporating topological patterns into dataflow frameworks for DSP system design by illustrating the pervasive nature of these patterns in the domain of DSP. We have already discussed a few such patterns in Chapter 1 — in

particular, the `butterfly` and `mesh` patterns, which have applications in FFTs and systolic arrays, respectively. Additionally, the `chain` pattern is one of the most commonly found topological patterns. This pattern finds applications in modeling multi-stage sample rate converters, delay lines in finite impulse response (FIR) filters, or configurations of pipeline stages. A chain of delay blocks, a chain of adders, and an `array` of filter taps collectively specify a complete FIR filter when connected together. A natural extension of this pattern is a 2-dimensional mesh structure. Such a structure is of particular use to model DSP architectures in which data flows across a network of processing elements connected to form a 2-D grid such as a systolic array, as discussed earlier in Chapter 1 [19].

A `ring` pattern represents a cycle in a graph as may be introduced by a phase-locked loop [22] or more generally, a `feedback loop` in the system. The FFT block is one of the most abundantly found blocks in DSP systems. An N-point FFT computation involves FFT computation stages of smaller dimensions that can be implemented as scaled versions of the 2-point FFT. These FFT stages resemble a butterfly-like pattern [24]. Such patterns can also be found in other applications, such as sorting networks [8]. Entropy encoding algorithms such as Huffman coding make use of the `binary tree` structure, a commonly found data structure in many computer algorithms [17]. A pattern in which edges connect a source node to multiple sink nodes can be termed as a `broadcast` pattern. This pattern finds use in applications that have computation blocks in multiple stages with blocks in one stage connected to those in the subsequent stage. Such patterns are observed in multi-layer neural networks used for pattern classification [10] and trellis coding algorithms used in digital communication [22]. It is also common to find its

dual, the `merge` pattern, which connects multiple source nodes to a single sink node. Applications may also have parallel connections between corresponding nodes in adjacent stages. We identify this pattern as a `parallel` pattern in which edges form a one-to-one correspondence between nodes in two different sets. We also identify a pattern called `multiedge` that creates multiple edges between a given pair of nodes.

3.2 Topological Patterns in DIF

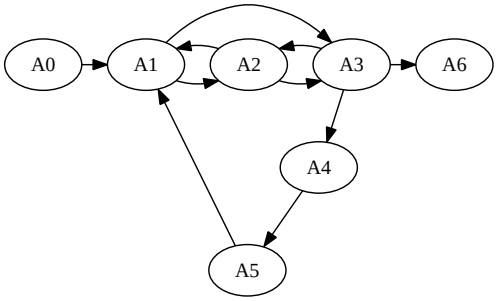
We extend the DIF language by supporting topological patterns as first class citizens in the modeling framework. These patterns can be defined as built-in patterns, which are recognized and processed through corresponding keywords in the language. To enable more flexible application of patterns, we also support declaring arbitrary (user-defined) patterns, whose associated graph construction functionality can be carried out through procedural language code (Java or C in the case of DIF) that is linked with the graph specification.

We have added, as built-in topological pattern specifiers, new keywords in DIF corresponding to topological patterns that are relatively common in signal processing systems. These keywords, such as `ring`, `parallel`, `merge`, `butterfly`, `broadcast`, and `chain`, allow specifying patterns explicitly as part of the `topology` block in a DIF specification. When declaring an instance of such a pattern, the designer must provide a sequence of vertices and an optional set of parameter values. The pattern construct, when parsed, generates the required edges, inserting the new edges into the graph that is being constructed. The pattern construct also configures the underlying nodes using the

parameter propagation mechanism explained in [28].

A typical way to specify a sequence of nodes is through the use of DIF notation for representing nodes in an array. For example, for an array of 7 nodes, specified as $A[7]$ (as in C, DIF arrays are indexed starting at 0), we can specify that 5 of its elements form a ring structure using the construct `ring(A[1:1:5])` in the `topology` block of the DIF code as shown in Fig. 3.1. The argument `A[1:1:5]` to the construct `ring`, specifies an array of nodes starting from `A[1]`, ending at `A[5]`, and having an array index increment of 1. Note that, outside of the pattern instantiation construct, the nodes in the array `A` can be accessed by their indices to create edges that are not part of the `ring` pattern. Thus, one can flexibly embed patterns within arbitrary structures including structures that contain other patterns.

It is also possible to generate multiple patterns that have one or more nodes common to them, as shown in Fig. 3.1. It is, thus, possible for the designer to effectively identify one or more types of overlapping topological patterns in the application graph.



(a)

```
topology {
    nodes = A[7];
    edges = e0(A[0], A[1]), e1(A[3], A[6]),
          ring_0[5] -> ring(A[1:1:5]),
          ring_1[3] -> ring(A[1], A[3], A[2]);
}
```

(b)

Figure 3.1: Overlapping patterns: (a) a graph topology having two `ring` patterns that have three nodes common to them, and (b) a corresponding DIF representation.

Chapter 4

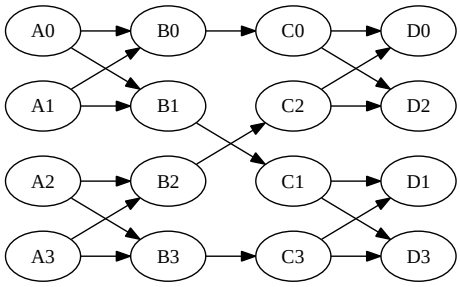
Applications of Topological Patterns

As described earlier, we envision topological patterns to offer a wide range of advantages at various stages of the design flow from modeling to platform-specific implementation. In Chapter 3, we have identified topological patterns in various DSP system specifications. In the following sections, we examine other aspects of the design flow where topological patterns can be effectively used.

4.1 Graph Analysis

The explicit specification of known graphical structures as topological patterns can significantly facilitate various types of dataflow graph analysis algorithms. For example, one of the first and most important steps in many dataflow graph scheduling strategies is to analyze the input graph to identify strongly connected components (SCCs). An SCC is a maximal subgraph in which every pair of distinct nodes is connected through a cyclic path. It is often useful to cluster SCCs — for example, SCCs can be clustered to improve scheduling of SDF graphs (e.g., see [16]). Such clustering of SCCs is typically performed in order to obtain a top-level *directed acyclic graph* (*DAG*). For a directed graph $G = (V, E)$, SCCs can be identified in $\Theta(|V| + |E|)$ time [8].

Consider an application graph that contains multiple feedback paths that can be modeled and specified using the `ring` pattern. A `ring` represents a cycle in the graph




```
topology {
    nodes = A[4], B[4], C[4], D[4];
    edges = fft2_0[4] -> butterfly(A[0:1], B[0:1]),
            fft2_1[4] -> butterfly(A[2:3], B[2:3]),
            fft4[8] -> butterfly(C[0:3], D[0:3]),
            e_par[4] -> parallel(B[0:3], C[0:3]);
}
```

Figure 4.1: Dataflow graph for a 4-point fast Fourier transform and the `topology` block in its DIF specification.

and hence, a subset of vertices that form an SCC. Such a cycle, when directly specified as a `ring` can be readily reduced into a single clustered actor. A `ring` with M nodes in it, when clustered into a single node, effectively reduces the number of nodes in the graph G by $M - 1$. Suppose that a graph G has many `ring` patterns that have been identified in the graph specification. Then by identifying these rings in constant time, which an analysis tool can do easily from explicit topological pattern specifications, the number of nodes and edges in the graph can be reduced significantly. This can lead to more efficient SCC computation, especially for large graphs.

4.2 Extracting Implementation-Specific Features

Fig. 4.1 shows an HSDF graph that models a 4-point FFT application [24], and the `topology` block in its DIF specification. Note the underlying topological patterns — `butterfly` and `parallel` — in the graph. It should also be noted that

`butterfly(C[0:3], D[0:3])` is a scaled version of a `butterfly` pattern with just 4 nodes, and is equivalent to two `butterfly` patterns formed by the node subsets {`C0, C2, D0, D2`} and {`C1, C3, D1, D3`}.

Apart from scalability, there is another useful feature in this HSDF graph representation. In particular, the bi-partite nature of both the patterns — `butterfly` and `parallel` — allows us to generate a pipelined implementation of this application. Here, segments A, B, C, and D, consisting of nodes `A[0:3]`, `B[0:3]`, `C[0:3]`, and `D[0:3]`, respectively, may be considered as pipeline stages of the FFT implementation. This inherent pipelined nature of the FFT application can be identified easily using the bi-partite nature of the underlying topological patterns. Of course, for FFTs, many efficient implementations have been developed in the literature, and the use of topological patterns does not add any obvious value to the large library of existing FFT implementation techniques. However, this example succinctly illustrates the general potential of topological patterns for exposing useful implementation options more clearly and efficiently to designers and to analysis modules within design tools.

4.3 Representing Schedules

The utility of topological patterns is not limited to representation of application graphs alone. Their utility can be extended to create concise and parameterizable representations of structures typical to schedules for certain application graphs. This can be of particular importance in functionally simulating application graphs, and porting schedules across design tools or languages. We elaborate on this using the following example.

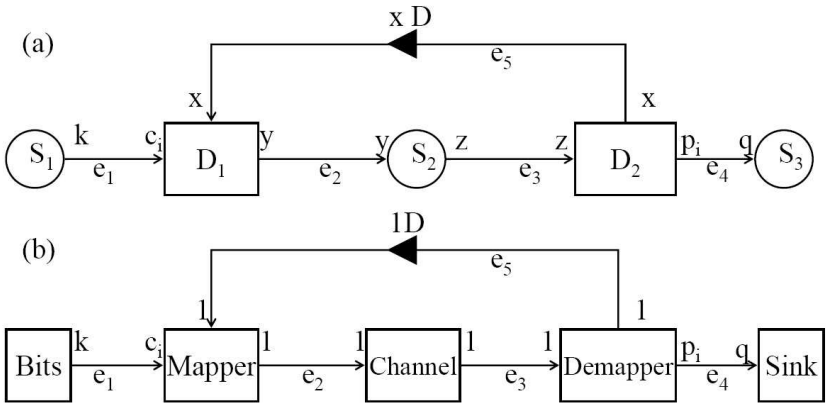
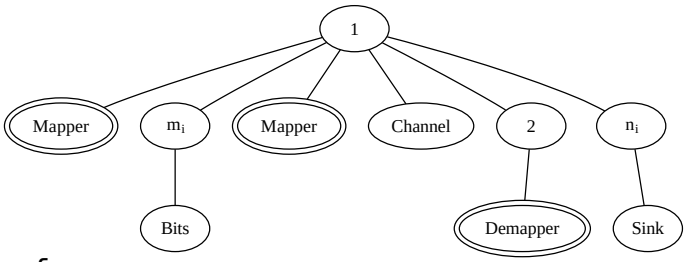


Figure 4.2: Dataflow graphs for (a) the generic class of applications under consideration, and (b) a simplified adaptive modulation scheme.

We consider a class of applications typically found in the domain of wireless communications, and signal processing systems that exhibit dataflow graph structures similar to the one shown in Fig. 4.2(a). A typical example of this type is that of the adaptive modulation scheme (*AMS*) shown in Fig. 4.2(b). The AMS is a dynamic communication application, which is an important part of modern wireless standards such as the *worldwide interoperability for microwave access* (*WiMAX*) [3] and *3rd generation partnership project — long term evolution* (*3GPP—LTE*) [1] standards. For details of AMS, we refer readers to [27]. There exist other applications that exhibit the general dataflow structure illustrated in Fig. 4.2(a), such as prediction error filters [14] and systems for frequency domain block adaptive filtering [30]. Such dataflow graphs can be efficiently simulated by constructing parameterized looped schedules (*PLS*s) as described in [27] and [18].

Fig. 4.3 shows a PLS for the AMS application. A PLS of this type is of particular importance since it can capture the dynamic dataflow behavior inherent in the application without compromising compile-time analysis. It is possible to perform useful analysis (e.g., estimation of upper bounds on total buffer memory requirements) for PLSs at




```
topology {
    nodes = Root, N[6], B, D, Snk;
    edges = e0[6] -> broadcast(Root, N[0:5]),
            e1(N[1], B), e2(N[4], D), e3(N[5], Snk);
}
```

Figure 4.3: A PLS for the application in Fig. 4.2(b), and the `topology` block in a corresponding DIF representation. Table 4.1 provides parameters associated with each `node` in the DIF specification.

compile-time.

In Fig. 4.2(a), the consumption rate c_i and production rate p_i can vary over finite ranges of positive integer values with known upper bounds $c_{\max}$ and $p_{\max}$, respectively. The subscript i in the symbols p_i and c_i represents the dependence of this production and consumption rate pair on the actor execution index i — thus, p_i represents the number of tokens produced onto e_4 in the ith execution (*firing*) of D_2, and c_i represents the number of tokens consumed from e_1 during the ith firing of D_1. In Fig. 4.3, the loop counts m_i and n_i are computed dynamically.

In the context of this AMS example, topological patterns help not only in specification of the application dataflow graph using the `ring` pattern, which can be used to identify the pair of dynamic actors easily, but also representation of generated PLSs using `broadcast` patterns with hierarchical nodes for SDF-schedules, as shown in Fig. 4.3. For such a well-structured schedule representation, it is possible to hand-tune an implementation and use that representation explicitly for applications having similar dataflow

Table 4.1: Actors and loop counts associated with nodes in the PLS graph representation. Here, NULL indicates an internal node in the GST that does not have any actor associated with it.

Node	Actor	Loop Count
Root	NULL	1
N[0]	Mapper	1
N[1]	NULL	m_i
N[2]	Mapper	1
N[3]	Channel	1
N[4]	NULL	2
N[5]	NULL	n_i
B	Bits	1
D	Demapper	1
Snk	Sink	1

Table 4.2: Average simulation times for different sink control conditions (numbers of tokens consumed by the sink) for the PLS in Fig. 4.3 using (1) GST traversal, and (2) a hand-tuned pattern-specific schedule.

Sink control condition	Average simulation time (ms)		Improvement
(Number of tokens)	(1)	(2)	(%)
10000	73	32	56.16
20000	90	47	47.78
50000	148	62	58.11
100000	248	93	62.50

behavior instead of traversing the GST using a generic process to derive a software or hardware implementation. In this case, topological patterns provide a framework by which hand-tuned schedules can be formally specified and reused across different applications or target platforms.

Table 4.2 shows a comparison between simulation times using GST traversal and hand-tuned pattern-specific implementation for the PLS in Fig. 4.3. These simulation experiments — the results of which are presented in Table 4.2 — differ from related experiments that we have reported on previously (e.g, in [29]) in that we have eliminated

some of the common overheads by suppressing printing of routine debug and status information. This allows us to determine the extent of effect of these two simulation strategies on simulation speed, and compare them more precisely. It can be seen that the hand-tuned software implementation results in faster simulations by a factor of up to 62%. Furthermore, through its formulation in the framework of topological patterns, the hand-tuned implementation can be analyzed, maintained, ported, and reused effectively across different design contexts.

4.4 Experimenting with Pattern-Specific Schedules

When specifying signal processing systems, an important motivation for using topological patterns is to facilitate application of pattern-specific transformations, such as pattern-specific scheduling transformations. In such a context, it can be useful for a design tool to provide features that allow the designer to experiment with various "scheduling patterns" at a high level of abstraction. Since topological patterns provide well-defined, scalable topological information, one can generate a structured schedule from a given pattern. We demonstrate this application of topological patterns through an example of a commonly used `butterfly` pattern.

Consider an SDF graph having a `butterfly` pattern, as shown in Fig. 4.4(a). One commonly used scheduling transformation involves applying clustering transformations on one pair of connected actors at a time such that no cycle is introduced in the resultant graph, and then generating a hierarchical schedule for the given application graph by iteratively applying such acyclic pairwise clustering (APC) [5]. In case of SDF graphs,

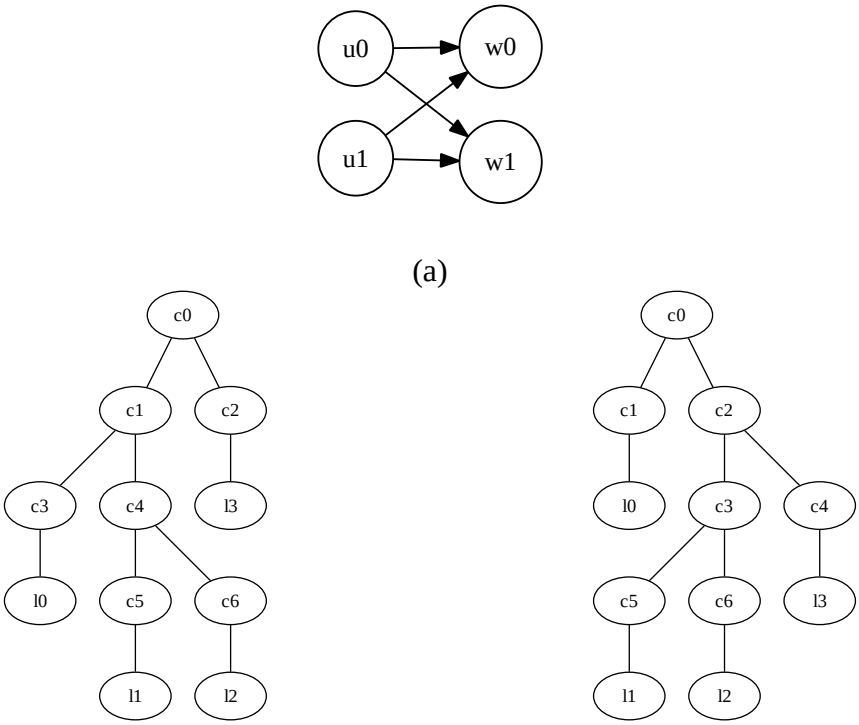


Figure 4.4: (a) An SDF graph with a `butterfly` pattern. (b)-(c) two possible GST structures using schedules that are based on acyclic pairwise clustering (iteratively clustering two actors at a time).

a group of actors can be *SDF-clustered* if its component actors can be scheduled together (i.e., the group can be scheduled as a single unit in the overall schedule for the graph) without introducing deadlock [5]. It can be observed that more than one schedule can be generated using APC depending on the pair of actors clustered at every stage of scheduling. In case of SDF graphs, the total buffer memory requirements depend upon the choice of a schedule, and in general, a schedule that has minimum total buffer memory requirements is desirable in many applications. A scheduling technique based on APC called acyclic pairwise grouping of adjacent nodes (APGAN) has been described in [5] that chooses a pair of actors to be clustered at every stage of scheduling using a metric based on repetition counts of the actors in the graph. This heuristic is widely used and attempts to minimize the total buffer memory requirements. We refer readers to [5] for more information on SDF-clustering, and SDF scheduling heuristics that are based on APC including APGAN.

A useful class of SDF schedules is that of single appearance looped schedules, as described in Section 2.1. Let $G(V, E)$ denote the graph in Fig. 4.4(a), where

$$V = \{u_0, u_1, w_o, w_1\}, \text{ and } E = \{(u_0, w_0), (u_0, w_1), (u_1, w_0), (u_1, w_1)\}, \tag{4.1}$$

and suppose that we apply APC to the graph. Based on the steps involved in APC, there are only two possible GST structures for this example. These two structures are shown in Fig. 4.4(b) and (c). Here, each c_i, $i = 0, 1, \cdots, 6$, denotes a loop count, while each l_i, $i = 0, 1, \cdots, 3$, denotes the actor pointed to by a leaf node in the GST. The existence of exactly two unique GST structures for this example can be verified from the following observations regarding the operation of APC (see [5] for further details on the operation

of APC for SDF graphs).

1. Let $U = \{u_0, u_1\}$, and $W = \{w_0, w_1\}$. Then we can describe the graph $G(V, E)$ as

$$V = U \cup W, \text{ and } E = U \times W. \tag{4.2}$$

2. Let $e \in E$ denote the group of actors clustered during the first clustering step. Then, $l_1 \in U$, and $l_2 \in W$. This follows from the bipartite nature of the `butterfly` pattern.

3. Following the first APC step, operation of APC ensures that $l_0 \in (U \setminus \{l_1\})$, and $l_3 \in (W \setminus \{l_2\})$. This is because clustering actors a and b such that $a \in U$ and $b \in W$ at this stage would amount to adding a cycle into the clustered graph, which is not permitted by APC.

4. Loop counts c_i, $i = 0, 1, \cdots, 6$, can be accordingly determined using the SDF repetitions vector (the vector of minimal repetition counts in a periodic schedule) for the application graph.

Given that each of the 4 pairs of actors can be grouped in the first-step, which, in turn, results in possibly two different schedules upon further grouping, we observe that there are at most 8 different single appearance looped schedules generated using this approach. Such different schedules can in general have different buffer memory requirements [5]. Thus, it can be useful for a designer to experiment with alternative schedules, estimate the buffer memory requirements for these schedules, and identify the schedule that best matches the application requirements and resource constraints.

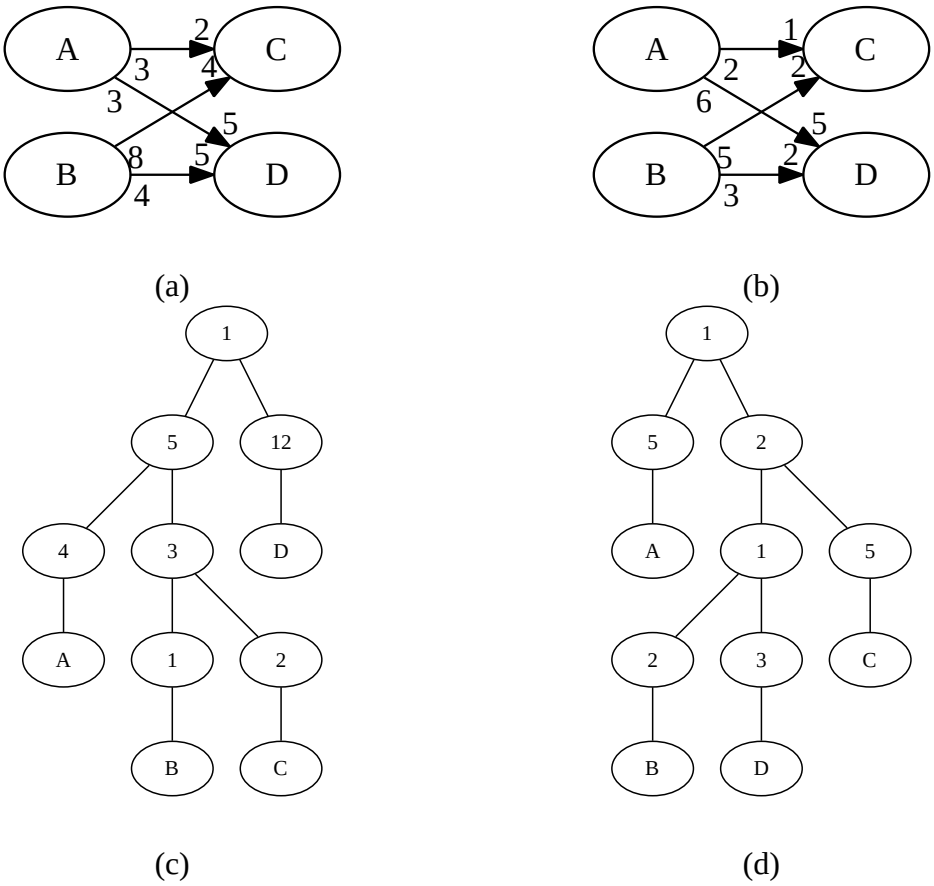


Figure 4.5: (a)-(b) SDF graphs with `butterfly` patterns. (c)-(d) GSTs for minimizing buffer memory requirements of the SDF graphs in (a) and (b), respectively.

Table 4.3: Buffer memory requirements for single appearance schedules generated from the SDF graph shown in Fig. 4.5(a).

Schedule	Single Appearance Schedule	Total buffer requirement (number of tokens)
Flat	(20 A)(15 B)(30 C)(12 D)	300
1	(5 (4 A)(3 B(2 C)))(12 D)	140
2	(20 A)(3 (5 B(2 C))(4 D))	148
3	(5 (3 B)(2 (2 A)(3 C)))(12 D)	150
4	(15 B)(2 (5 (2 A)(3 C))(6 D))	216
5	(15 B)(4 (5 A)(3 D))(30 C)	255
6	(15 B)(2 (2 (5 A)(3 D))(15 C))	225
7	(20 A)(3 (5 B)(4 D))(30 C)	260
8	(20 A) (3 (5 B)(4 D)(10 C))	180

Table 4.4: Buffer memory requirements for single appearance schedules generated from the SDF graph shown in Fig. 4.5(b).

Schedule	Single Appearance Schedule	Total buffer requirement (number of tokens)
Flat	(5 A)(4 B)(10 C)(6 D)	72
1	(4 B)(5 A(2 C))(6 D)	64
2	(5 A)(2 (2 B)(5 C)(3 D))	56
3	(5 A)(2 (2 B)(5 C))(6 D)	62
4	(5 A)(2 (2 B)(3 D))(10 C)	66
5	(5 A)(2 (2 B)(3 D)(5 C))	56

For the `butterfly` pattern shown in Fig. 4.5(a), Table 4.3 shows 9 different schedules, including a flat schedule for comparison. It can be seen that each of these schedules has different buffer memory requirements. In a given design context, a designer may want to experiment with all schedules that fit the available resources in the target platform. The optimal schedule from the viewpoint of total buffer memory cost (schedule (1)) has a total buffer memory cost of 140 memory units, and is generated using the APGAN strategy.

However, APGAN is in general a heuristic and is therefore not always guaranteed to derive an optimal solution. For example, consider the `butterfly` pattern shown in Fig. 4.5(b). Table 4.4 shows 6 different schedules for this graph, including, again, a flat schedule, and 5 different looped schedules. Here, schedule (1) is the one generated by applying the APGAN strategy, and it can be seen that schedules (2), (3), and (5) outperform this schedule in terms of total buffer memory requirements.

This example demonstrates the utility of experimenting with alternative schedules even if established heuristics, such as APGAN, are available. Topological patterns facil-

itate such experimentation through their capabilities for schedule representation. In particular, topological patterns allow designers to construct structured patterns of schedules, which can then be examined separately to determine which one is most suitable in a given design context. Furthermore, topological pattern representations can be used to maintain libraries of subsystem-specific schedules, which can then be drawn upon efficiently when constructing larger applications that employ those subsystems.

Chapter 5

Summary and Conclusions

We have introduced the concept of topological patterns, which can be used to identify and concisely iterate across arbitrary structures in a dataflow application graph. We have shown how the types of flowgraph substructures that are pervasive in the DSP application domain can be effectively represented in terms of topological patterns, and thereby used to generate compact, scalable application representations.

We have also shown how an underlying design tool can exploit a high-level application specification consisting of topological patterns in various aspects of the design flow. In particular, we have demonstrated the efficacy of topological patterns in dataflow graph analysis, and extracting implementation-specific features. We have applied the concept of topological patterns to represent schedules for application graphs. Such representations are useful, for example, when porting schedules generated using one design tool to other platform-specific tools or design languages. We have demonstrated the utility of experimentation with pattern-specific scheduling transformations, and how topological patterns facilitate such experimentation.

Useful directions for further investigation include automating the application and integration of topological patterns and analysis techniques that are driven by specific topological patterns.

Bibliography

[1] 3GPP TS 36.211 V8.7.0 (2009-05): Physical channels and modulation (2009)

[2] Andrade, H., Kovner, S.: Software synthesis from dataflow models for G and LabVIEW^{TM}. In: Signals, Systems Computers, 1998. Conference Record of the Thirty-Second Asilomar Conference on, vol. 2, pp. 1705 –1709 vol.2 (1998). DOI 10.1109/ACSSC.1998.751616

[3] Andrews, J.G., Ghosh, A., Muhamed, R.: Fundamentals of WiMAX: understanding broadband wireless networking. Prentice Hall (2007)

[4] Bhattacharya, B., Bhattacharyya, S.S.: Parameterized dataflow modeling of DSP systems. In: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, pp. 1948–1951. Istanbul, Turkey (2000)

[5] Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: Software Synthesis from Dataflow Graphs. Kluwer Academic Publishers (1996)

[6] Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.A.: Cyclo-static dataflow. IEEE Transactions on Signal Processing **44**(2), 397–408 (1996)

[7] Buck, J.T.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, EECS Department, University of California, Berkeley (1993). URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/1993/2429.html`

[8] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, second edn. MIT Press and McGraw-Hill (2001)

[9] Corretjer, I., Hsu, C., Bhattacharyya, S.S.: Configuration and representation of large-scale dataflow graphs using the dataflow interchange format. In: Proceedings of the IEEE Workshop on Signal Processing Systems, pp. 10–15. Banff, Canada (2006)

[10] Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification, second edn. John Wiley and Sons, Inc. (2000)

[11] Eker, J., Janneck, J.W.: CAL language report, language version 1.0 — document edition 1. Tech. Rep. UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley (2003)

[12] Gamma, E., Helm, R., Johnson, R., Vissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)

[13] Haubelt, C., Falk, J., Keinert, J., Schlichter, T., Streubhr, M., Deyhle, A., Hadert, A., Teich, J.: A SystemC-based design methodology for digital signal processing systems. EURASIP Journal on Embedded Systems **2007**, Article ID 47,580, 22 pages (2007)

[14] Haykin, S.: Adaptive filter theory. Prentice-Hall, Inc. (1996)

[15] Hsu, C., Ko, M., Bhattacharyya, S.S.: Software synthesis from the dataflow interchange format. In: Proceedings of the International Workshop on Software and Compilers for Embedded Systems, pp. 37–49. Dallas, Texas (2005)

[16] Hsu, C., Ramasubbu, S., Ko, M., Pino, J.L., Bhattacharyya, S.S.: Efficient simulation of critical synchronous dataflow graphs. In: Proceedings of the Design Automation Conference, pp. 893–898. San Francisco, California (2006)

[17] Huffman, D.A.: A method for the construction of minimum-redundancy codes. In: Proceedings of the IRE, pp. 1098–1101 (1952)

[18] Ko, M., Zissulescu, C., Puthenpurayil, S., Bhattacharyya, S.S., Kienhuis, B., Deprettere, E.: Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. IEEE Transactions on Signal Processing **55**(6), 3126–3138 (2007)

[19] Kung, S.Y.: VLSI Array processors. Prentice Hall (1988)

[20] Lee, E.A.: Recurrences, iteration, and conditionals in statically scheduled block diagram languages. In: Proceedings of the International Workshop on VLSI Signal Processing (1988)

[21] Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous dataflow programs for digital signal processing. IEEE Transactions on Computers **C-36**(1), 24–35 (1987). DOI 10.1109/TC.1987.5009446

[22] Lee, E.A., Messerschmitt, D.G.: Digital Communication. Kluwer Academic Publishers (1988)

[23] Lee, E.A., Parks, T.M.: Dataflow process networks. Proceedings of the IEEE pp. 773–799 (1995)

[24] Oppenheim, A.V., Schafer, R.W.: Discrete-Time Signal Processing. Prentice-Hall, Inc. (1989)

[25] Pino, J.L., Kalbasi, K.: Cosimulating synchronous DSP applications with analog RF circuits. In: Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers (1998)

[26] Plishker, W., Sane, N., Kiemb, M., Anand, K., Bhattacharyya, S.S.: Functional DIF for rapid prototyping. In: Proceedings of the International Symposium on Rapid System Prototyping, pp. 17–23. Monterey, California (2008)

[27] Sane, N., Hsu, C., Pino, J.L., Bhattacharyya, S.S.: Simulating dynamic communication systems using the core functional dataflow model. In: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, pp. 1538–1541. Dallas, Texas (2010)

[28] Sane, N., Kee, H., Seetharaman, G., Bhattacharyya, S.S.: Scalable representation of dataflow graph structures using topological patterns. In: Proceedings of the IEEE Workshop on Signal Processing Systems, pp. 13–18. San Francisco Bay Area, USA (2010)

[29] Sane, N., Kee, H., Seetharaman, G., Bhattacharyya, S.S.: Topological patterns for scalable representation and analysis of dataflow graphs. Journal of Signal Processing Systems — Special Issue on SiPS 2010 (2011). DOI 10.1007/s11265-011-0610-1

[30] Shynk, J.J.: Frequency-domain and multirate adaptive filtering. IEEE Signal Processing Magazine **9**(1), 14–37 (1992)

[31] Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., Deprettere, E.: System design using Kahn process networks: the Compaan/Laura approach. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, vol. 1, pp. 340 – 345 Vol.1 (2004). DOI 10.1109/DATE.2004.1268870

[32] Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: International Conference on Compiler Construction. Grenoble, France (2002)

[33] Verbauwhede, I.M., Scheers, C.J., Rabaey, J.M.: Specification and support for multidimensional DSP in the SILAGE language. In: IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 2, pp. II/473 –II/476 (1994). DOI 10.1109/ICASSP.1994.389622