In Journal of Signal Processing Systems, 65(2):229-244, 2011.

Topological patterns for scalable representation and analysis of dataflow graphs

Nimish Sane · Hojin Kee · Gunasekaran Seetharaman · Shuvra S. Bhattacharyya

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 DSP specification languages and design tools, we develop the modeling concept of *topological patterns*, and propose ways for supporting this concept in a high-level language. We augment the dataflow interchange format (DIF) language — a language for specifying DSP-oriented dataflow graphs — with constructs for supporting topological patterns, and we show how topological patterns can be effective in various aspects of embedded signal processing design flows using specific application examples.

Keywords Dataflow graphs · High-level languages · Model-based design · Topological patterns · Signal processing systems

N. Sane
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies,
University of Maryland, College Park, MD 20742, USA.
Tel.: +1 301-405-8295
E-mail: nsane@umd.edu

H. Kee
National Instruments,
Austin, TX 78759, USA.

G. Seetharaman
Air Force Research Laboratory,
Rome, NY, USA.

S. S. Bhattacharyya
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies,
University of Maryland, College Park, MD 20742, USA.

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 [28], National Instruments LabVIEW [2], Compaan/Laura [34], and SysteMoc [15]. 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 paper, 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* [26], 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 [21]. 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 [17], [29].

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 paper, we provide background on dataflow modeling and the DIF language in Section 2, and discuss relevant prior work in Section 3. We elaborate the concept of topological patterns in Section 4. In Section 5, we describe how we extend the DIF language to integrate topological patterns as a first class modeling construct. In Section 6, 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; representing equivalent homogeneous synchronous dataflow (HSDF) graphs of application graphs modeled using synchronous dataflow (SDF) and cyclostatic dataflow (CSDF) models; extracting implementation-specific features; exploring trade-offs for an FPGA implementation of a JPEG image compression application; representing schedules; and experimenting with pattern-specific schedules. We conclude in Section 7.

2 Background

This section 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*, $\text{del}(e)$, 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 SDF, these numbers are constant across all actor firings for a given input or output [23]. 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, $\text{src}(e)$ and $\text{snk}(e)$ denote the source actor and sink actor of the edge, respectively. Additionally, if e is an SDF edge, then $\text{prd}(e)$ represents the number of tokens produced on the edge by each firing of $\text{src}(e)$, while $\text{cns}(e)$ represents the number of tokens consumed from the edge by each firing of $\text{snk}(e)$.

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 [23]; periodic patterns of constant values as in CSDF [7]; or more complex forms that are data-dependent (e.g., see [8], [4], and [29]).

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 paper, 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

$$q(\text{src}(e)) \times \text{prd}(e) = q(\text{snk}(e)) \times \text{cns}(e) \tag{1}$$

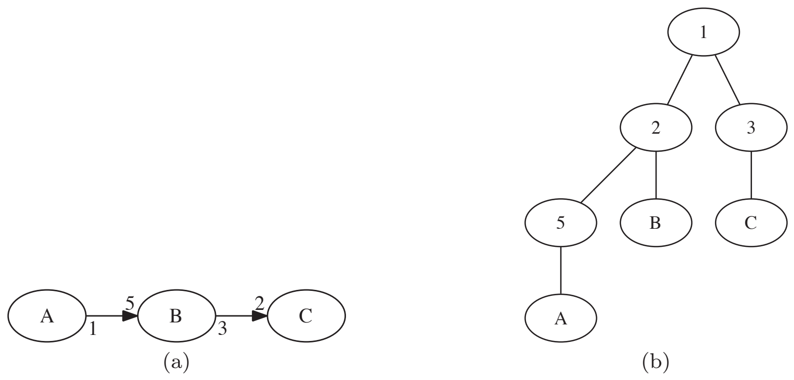


Fig. 1 (a) An SDF graph for a sample rate converter. (b) A schedule for this graph that is represented using a GST.

for each edge $e \in E$ [23].

For example, consider the SDF graph shown in Fig. 1a. The repetition counts 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. 1a 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. 1a 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 [20]. 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 count(u) and 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 sub-schedule rooted at any internal node is executed as many times as specified by the loop count of that node. Figure 1b shows a GST for a valid looped schedule for the SDF graph shown in Fig. 1a. This particular GST represents the firing sequence $(2\ (5\ A)B)(3\ C)$.

```
[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, ...;
    }
}
```

Fig. 2 The DIF language

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 [17]. 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.

Figure 2 illustrates some of the available constructs in the DIF language along with the syntax used for application specification. More details on the DIF language can be found in [17]. 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 comprehensive functional simulation framework, and a software synthesis framework for generating C code [17], [29]. 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.

3 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 [22]. Subsequent work includes evolution of various textual languages for DSP system design, such as SILAGE [37], StreamIt [36], and CAL [12]. 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 [25]. 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). 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 development 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 [10]. 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) 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 can be accessed using its index as `tap[i-1]`. 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.

In this paper, we formulate the concept of topological patterns and its application to dataflow modeling, and 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.

A preliminary version of this work was presented in [31]. This paper goes beyond the developments of [31] by significantly extending the development of applications of 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.

4 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 [13], 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.

4.1 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 Section 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 Section 1 [21].

A `ring` pattern represents a cycle in a graph as may be introduced by a phase-locked loop [24] 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 [26]. Such patterns can also be found in other applications, such as sorting networks [9]. Entropy encoding algorithms such as Huffman coding make use of the `binary tree` structure, a commonly found data structure in many computer algorithms [19]. 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 [11] and trellis coding algorithms used in digital communication [24]. 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.

4.2 Parameter Propagation

An important feature to support in conjunction with topological patterns is a mechanism for structured *parameter propagation*, whereby any parameters associated with the vertices in a topological pattern can be set as a function of the vertex indices (i.e., indices associated with the underlying vertex ordering that is input to the pattern instance). For example, Fig. 3 shows an array of 5 actors identified as $A\_1, A\_2, \ldots, A\_5$, where each actor has a parameter *angle* associated with it that is an affine function of its index in the array. Such a parameter assignment

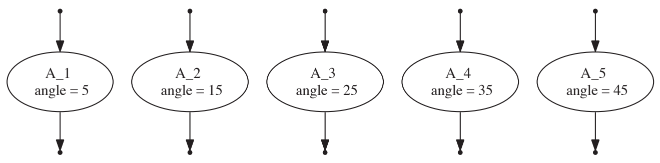


Fig. 3 Configuring an array of nodes with a PPP.

can be implemented in a scalable, reusable, and explicitly-recognizable form as a designated PPP — in particular, a PPP for affine mappings of parameter values across ordered vertices. Such an affine PPP can find use in specifying elements of a *steering vector* corresponding to each sensor in a sensor array while estimating the direction of arrival of the received signal [16].

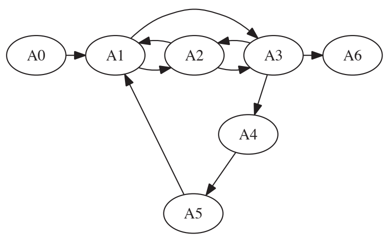
One of our important motivations for using topological patterns is to provide for compact, scalable representations for large dataflow graphs. It is common for such large graphs to have actors with the same functionality that scale in number with the size of the application graph. These actors may have functional parameters (for example, the parameter *angle* associated with the actors in Fig. 3) that determine some of their functional aspects and also distinguish them from the other actors of the same (parameterized) functionality. It may be inconvenient to specify such parameters individually for all of the actors with growing size of an application graph (in fact, such individualized parameter specification violates the compactness objective of topological patterns). PPPs can help here by providing a compact representation format that can be used to set parameters associated with actors in the large graphs that are represented by the associated topological patterns.

In terms of implementation in the DIF language, just as component attributes and topological patterns can be either user-defined or built-in, similarly commonly-used PPPs can be absorbed into the language as built-in PPPs, while users have the flexibility to incorporate specialized PPPs by linking their interpretation (propagation functionality) to segments of customized Java code.

5 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



(a)

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

(b)

Fig. 4 Overlapping patterns: (a) a graph topology having two `ring` patterns that have three nodes common to them, and (b) a corresponding DIF representation.

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 Section 4.2.

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. 4. 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. 4. It is, thus, possible for the designer to effectively identify one or more types of overlapping topological patterns in the application graph.

6 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 Sections 4 and 5, we have identified topological patterns in various DSP system specifications. In the following subsections, we examine other aspects of the design flow where topological patterns can be effectively used.

6.1 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 [18]). 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 [9].

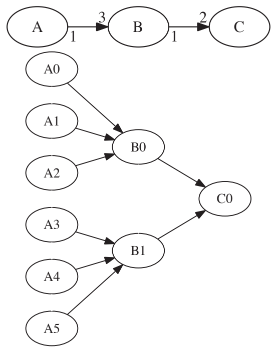
Consider an application graph that contains multiple feedback paths that can be modeled and specified using the `ring` pattern. A `ring` represents a cycle in the graph and hence, a subset of vertices that form an SCC. Such a cycle, when directly specified as a `ring` can be readily reduced into a single 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.

6.2 Representing HSDF Graphs

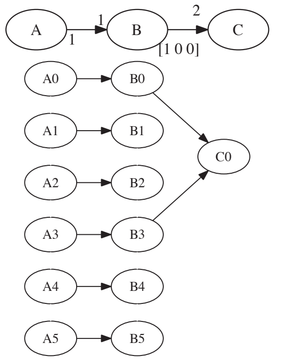
Many techniques devised for generating multiprocessor schedules from SDF graphs require that the given dataflow graph be transformed into an equivalent HSDF graph (e.g., see [33]). An HSDF graph is an SDF graph in which every actor consumes (produces) a single token from (on) each input (output) port. Techniques for converting an SDF graph into equivalent (for scheduling purposes) HSDF graphs have been developed in [23]. Such techniques are useful, because equivalent HSDF graphs can expose parallelism much more effectively compared to their more compact SDF counterparts.

Unfortunately, equivalent HSDF representations can scale very inefficiently — the size of an equivalent HSDF graph is in general not polynomially bounded in the size of the corresponding SDF graph [27]. Representing such HSDF graphs becomes a cumbersome exercise, as such representations require large amounts of storage to maintain and large amounts of computation time to process them. For a large HSDF representation, it is difficult for a design tool to traverse the HSDF representation and make effective use of it within a reasonable amount of time. Topological patterns can help in this situation by providing concise representations to expose repetitive structures within HSDF representations, thereby improving the efficiency of HSDF-based schedulers.

For example, Fig. 5a shows an SDF graph that models a simple sample rate converter, and its equivalent HSDF graph (below). Here, actor B is a decimator with a decimation factor of 3. Fig. 5c shows a DIF specification of this HSDF graph using topological patterns. Figure 5b and d show an equivalent CSDF graph model



(a) SDF graph to HSDF graph



(b) CSDF graph to HSDF graph

```
topology {
    nodes = A[6], B[2], C;
    edges = merge(A[0:2], B[0]), merge(A[3:5], B[1]),
            merge(B[0:1], C);
}
```

(c) DIF `topology` block for HSDF graph in (a)

```
topology {
    nodes = A[6], B[6], C;
    edges = parallel(A[0:5], B[0:5]),
            merge(B[0:3:3], C);
}
```

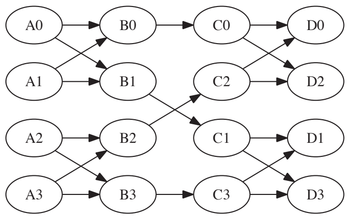
(d) DIF `topology` block for HSDF graph in (b)

Fig. 5 A sample rate converter.

with its HSDF graph and associated topological-pattern-based DIF specification. In the CSDF representation, actor B provides a decimation by a factor of 3. Actor B consumes input tokens on every firing while producing an output token only on every third firing, starting with the first firing. As this example illustrates, topological patterns can provide a concise and scalable representation of equivalent HSDF graph representations for SDF and CSDF graphs.

It should, however, be noted that a graph representation using topological patterns is in general not unique. Depending on the set of available topological patterns, it may be possible to have multiple functionally-equivalent representations of a given dataflow graph using topological patterns. In the case of Fig. 5a, for example, it may be possible to use a `tree` pattern if the associated design tool supports it.

Structured representations of HSDF graphs can also enable efficient tuning of HSDF graph representations in terms of application parameters. For example, for the dataflow graph in Fig. 5b, it can be observed that if the decimation factor of actor B is changed, then the DIF representation for the HSDF graph can be updated by simply changing the numeric arguments to the topological patterns used in its representation. In general, for a decimation factor of M, the production rate of actor B in Fig. 5b is $[1\ 0\ 0 \cdots 0]_{1\times M}$ and the equivalent HSDF graph for this CSDF graph has the following specification, where M is a suitably-declared parameter.



```
topology {
    nodes = A[4], B[4], C[4], D[4];
    edges = butterfly(A[0:1], B[0:1]), butterfly(A[2:3], B[2:3]),
            butterfly(C[0:3], D[0:3]), parallel(B[0:3], C[0:3]);
}
```

Fig. 6 Dataflow graph for a 4-point fast Fourier transform and the `topology` block in its DIF specification.

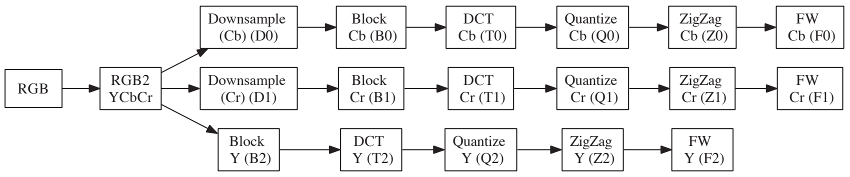
```
topology {
    nodes = A[2M], B[2M], C;
    edges = parallel(A[0:2M-1], B[0:2M-1]),
            merge(B[0:M:M], C);
}
```

Thus, topological patterns provide streamlined representations that are concise, tunable, and scalable, and are particularly useful for complex graph structures, such as those found in equivalent HSDF graphs arising from multirate SDF, and CSDF models.

6.3 Extracting Implementation-Specific Features

Figure 6 shows an HSDF graph that models a 4-point FFT application [26], 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.



```
topology {
    nodes = RGB, RGB2YCbCr, D[2], B[3], T[3], Q[3], Z[3], F[3];
    edges = e0(RGB, RGB2YCbCr), broadcast(RGB2YCbCr, D[0:1:1], B[2]),
            chain(D[0], B[0], T[0], Q[0], Z[0], F[0]),
            chain(D[1], B[1], T[1], Q[1], Z[1], F[1]),
            chain(B[2], T[2], Q[2], Z[2], F[2]);
}
```

Fig. 7 JPEG encoder and the `topology` block in its DIF specification.

6.4 Exploring Implementation Trade-offs

Figure 7 shows a JPEG encoder along with the `topology` block in its DIF specification [38]. It effectively employs the `broadcast` and `chain` patterns in its representation. The JPEG compression algorithm downsamples both the chroma (C_b and C_r) components before processing them. Except for this, all three components (both chroma and luma Y) are processed through functionally similar chains of blocks. The input pixels are grouped into blocks that are then transformed using the discrete cosine transform (DCT), quantized, and scanned in a zigzag order. In particular, the chroma components may be processed using shared functional modules that are clearly exposed by the `topology` block. Without the use of topological patterns, this observation may not be clear to a designer until the entire graph is carefully traced. For a design tool, this observation may go entirely unexploited because such high level structure can be difficult to extract automatically from unstructured specifications.

The problem of identifying such graph structure is related to the *graph isomorphism problem*, which is the problem of detecting whether two graphs (or two subgraphs from the same or different graphs) can have their vertices and edges placed in one-to-one correspondence with one another in a manner that maintains edge-vertex connectivity relationships. There are no known polynomial time algorithms for the graph isomorphism problem (e.g., see [14]).

For the JPEG encoder example, we can exploit the potential for resource sharing — which is exposed explicitly at a high-level through the use of topological patterns — to develop a streamlined FPGA implementation. Awareness of the high level topological pattern in this application allows for systematic trade-off analysis between two design options — one with shared resources for chroma component processing, and another without shared resources.

An analysis of the high level dataflow specification suggests that downsampling of chroma components would ensure that the chain processing Y component is the bottleneck and hence, the throughput should remain unaffected even when the C_b and C_r components are processed using shared functional modules. Precise modeling of the shared-resource implementation of the JPEG encoder requires that the SDF design in Fig. 7 be transformed to expose more detail. For example, the design can be converted into an equivalent CSDF design in which buffers between

Table 1 Performance and resource utilization trade-offs for FPGA implementation of a JPEG encoder.

JPEG Encoder	Throughput (samples /cycle)	FPGA Resource Utilization		
		Slices (out of 13696)	18kB BRAM	18x18 MULT
Non-shared	0.159	8070 (58%)	41	30
Shared	0.159	6088 (44%)	37	22

functional modules are duplicated and alternate buffers are used in successive schedule iterations. For more background on this form of CSDF-based structural modeling, we refer the reader to [5].

From inspection of the CSDF intermediate model, it can be reasoned that the buffer requirement would remain unchanged across both designs (shared- versus separate-resource). However, we expect that the shared-resource version of the JPEG encoder would result in a net reduction in BRAM utilization.

This analysis can be confirmed from the resource utilization and throughput for shared- and separate-resource JPEG encoder implementations on the Xilinx Virtex-II Pro FPGA, as shown in Table 1 [31]. The base clock rate for our experiments is 40 MHz. Even though actor-level resource sharing is often avoided in FPGA implementation due to the relatively high costs of multiplexing and routing resources (e.g., see [35]), resource sharing for a subgraph in a dataflow representation can result in conservation of FPGA resources that overrides the multiplexing overhead. The shared-resource JPEG encoder uses less BRAM than the separate-resource version, which can be attributed to the shared DCT block. Also, the shared-resource version uses fewer (18×18) multiplier units by employing shared downsampling, DCT, and quantization modules.

As expected — from the aforementioned bottleneck analysis — both versions of the JPEG encoder achieve the same throughput. In particular, the Y component remains as the system bottleneck even when the C_b and C_r components are processed using shared FPGA resources. Our experiments thus demonstrate concretely how topological patterns can provide a *formal path* from scalable application analysis to the systematic exploration of implementation trade-offs in the design and implementation of signal processing systems on a relevant target platform.

6.5 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.

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. 8a. A typical example of this type is that of

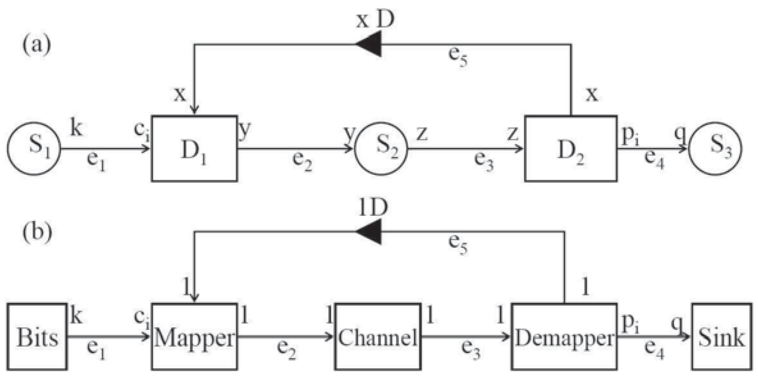


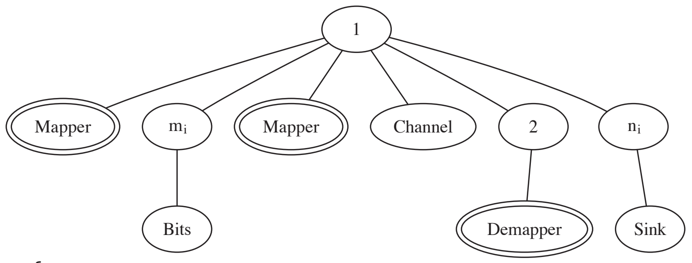
Fig. 8 Dataflow graphs for (a) the generic class of applications under consideration, and (b) a simplified adaptive modulation scheme.

the adaptive modulation scheme (*AMS*) shown in Fig. 8b. 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 [30]. There exist other applications that exhibit the general dataflow structure illustrated in Fig. 8a, such as prediction error filters [16] and systems for frequency domain block adaptive filtering [32]. Such dataflow graphs can be efficiently simulated by constructing parameterized looped schedules (*PLS*s) as described in [30] and [20].

Figure 9 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 compile-time.

In Fig. 8a, 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. 9, 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. 9. 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 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.



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

Fig. 9 A PLS for the application in Fig. 8b, and the `topology` block in a corresponding DIF representation. Table 2 provides parameters associated with each `node` in the DIF specification.

Table 2 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 3 Average simulation times for different sink control conditions (numbers of tokens consumed by the sink) for the PLS in Fig. 9 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)	(%)
100	568	399	29.75
500	2337	2015	13.78
1000	4661	4099	13.99
5000	23537	20426	13.22

Table 3 shows a comparison between simulation times using GST traversal and hand-tuned pattern-specific implementation for the PLS in Fig. 9. It can be seen that the hand-tuned software implementation results in faster simulations by a factor of up to 29%. 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.

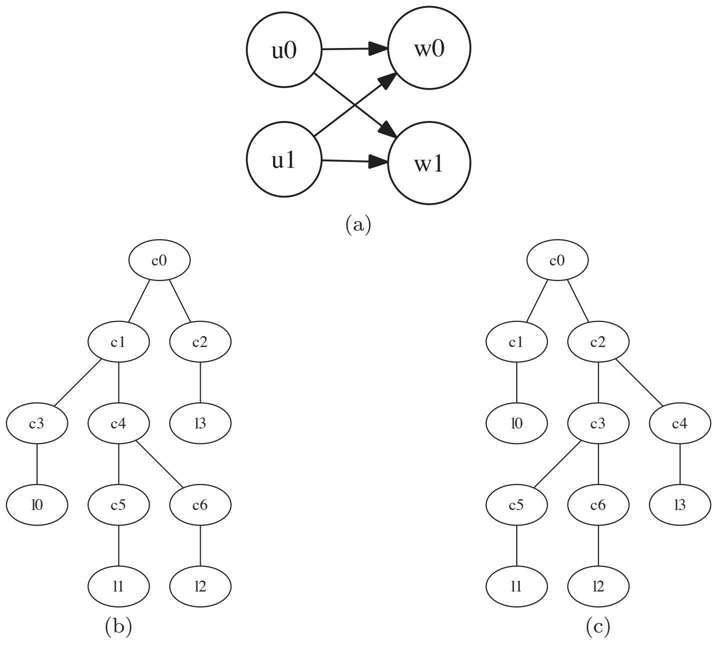


Fig. 10 (a) An SDF graph with a `butterfly` pattern. (b)-(c) two possible GST structures using schedules that are based on acyclic pairwise clustering (iteratively clustering two actors at a time).

6.6 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. 10a. 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) [6]. In case of SDF graphs, 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 [6]. 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 [6] 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 [6] 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. 10a, where

$$V = \{u_0, u_1, w_o, w_1\}, \text{ and } E = \{(u_0, w_0), (u_0, w_1), (u_1, w_0), (u_1, w_1)\}, \quad (2)$$

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. 10b 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 [6] 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
$$V = U \cup W, \text{ and } E = U \times W. \quad (3)$$
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 [6]. 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.

For the `butterfly` pattern shown in Fig. 11a, Table 4 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.

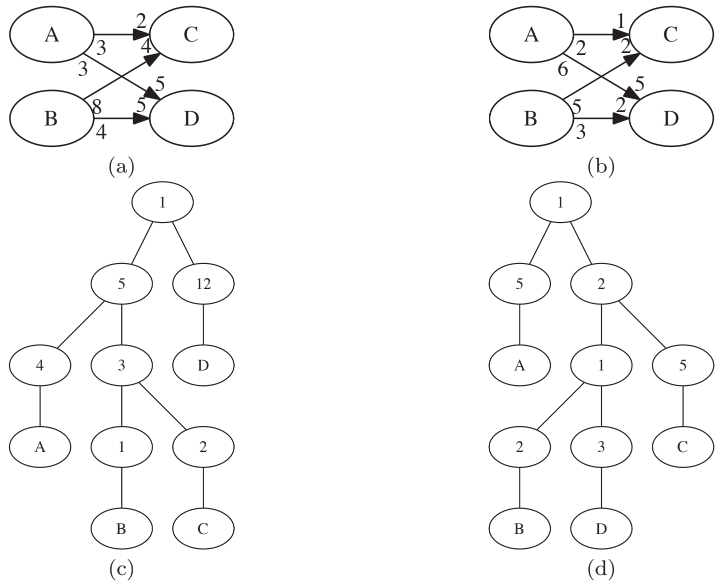


Fig. 11 (a)-(b) SDF graphs with `butterfly` patterns. (c)-(d) GSTs for minimizing buffer memory requirements of the SDF graphs in (a) and (b), respectively.

Table 4 Buffer memory requirements for single appearance schedules generated from the SDF graph shown in Fig. 11a.

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 5 Buffer memory requirements for single appearance schedules generated from the SDF graph shown in Fig. 11b.

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

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. 11b. Table 5 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 facilitate 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.

7 Conclusion

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, concise and scalable representation of HSDF graphs, and exploring implementation-specific trade-offs. We have also shown the use of topological patterns in 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.

Acknowledgements Authors acknowledge Government's support in publication of this paper. The material is based upon work funded by the US Air Force Research Laboratory (AFRL-FA87501110049), Laboratory for Telecommunication Sciences, National Radio Astronomy Observatory, US National Science Foundation, and Austrian Marshall Plan Foundation. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not reflect the views of AFRL, LTS, NRAO, NSF, or AMPF.

References

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., Leupers, R., Marwedel, P.: Software synthesis and code generation for DSP. IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing **47**(9), 849–875 (2000)
6. Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: Software Synthesis from Dataflow Graphs. Kluwer Academic Publishers (1996)
7. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.A.: Cyclo-static dataflow. IEEE Transactions on Signal Processing **44**(2), 397–408 (1996)
8. 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`
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms, second edn. MIT Press and McGraw-Hill (2001)
10. 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)
11. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern classification, second edn. John Wiley and Sons, Inc. (2000)
12. 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)
13. Gamma, E., Helm, R., Johnson, R., Vissides, J.: Design patterns: Elements of reusable object-oriented software. Addison-Wesley (1995)
14. Garey, M.R., Johnson, D.S.: Computers and intractability: A guide to the theory of NP-Completeness. W. H. Freeman and Company (1979)
15. 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)
16. Haykin, S.: Adaptive filter theory. Prentice-Hall, Inc. (1996)
17. 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)
18. 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)
19. Huffman, D.A.: A method for the construction of minimum-redundancy codes. In: Proceedings of the IRE, pp. 1098–1101 (1952)
20. 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)
21. Kung, S.Y.: VLSI array processors. Prentice Hall (1988)
22. Lee, E.A.: Recurrences, iteration, and conditionals in statically scheduled block diagram languages. In: Proceedings of the International Workshop on VLSI Signal Processing (1988)
23. 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
24. Lee, E.A., Messerschmitt, D.G.: Digital communication. Kluwer Academic Publishers (1988)

25. Lee, E.A., Parks, T.M.: Dataflow process networks. Proceedings of the IEEE pp. 773–799 (1995)
26. Oppenheim, A.V., Schafer, R.W.: Discrete-time signal processing. Prentice-Hall, Inc. (1989)
27. Pino, J.L., Bhattacharyya, S.S., Lee, E.A.: A hierarchical multiprocessor scheduling system for DSP applications. In: Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers, pp. 122–126 vol.1. Pacific Grove, California (1995)
28. 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, vol. 2, pp. 1710–1714. Pacific Grove, California (1998)
29. 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)
30. 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)
31. 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. San Francisco Bay Area, USA (2010)
32. Shynk, J.J.: Frequency-domain and multirate adaptive filtering. IEEE Signal Processing Magazine **9**(1), 14–37 (1992)
33. Sriram, S., Bhattacharyya, S.S.: Embedded multiprocessors: Scheduling and synchronization, second edn. CRC Press (2009)
34. 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
35. Sun, W., Wirthlin, M.J., Neuendorffer, S.: FPGA pipeline synthesis design exploration using module selection and resource sharing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **26**(2), 254–265 (2007)
36. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: International Conference on Compiler Construction. Grenoble, France (2002)
37. 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
38. Wallace, G.K.: The JPEG still picture compression standard. IEEE Transactions on Consumer Electronics **38**(1), xviii –xxxiv (1992). DOI 10.1109/30.125072