In Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece, July 2004. To appear.

DIF: An Interchange Format for Dataflow-based Design Tools

Chia-Jui Hsu, Fuat Keceli, Ming-Yung Ko, Shahrooz Shahparnia, and Shuvra S. Bhattacharyya

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

Abstract. The dataflow interchange format (DIF) is a textual language that is geared towards capturing the semantics of graphical design tools for DSP system design. A key objective of DIF is to facilitate technology transfer across dataflow-based DSP design tools by providing a common, extensible semantics for representing coarse-grain dataflow graphs, and recognizing useful sub-classes of dataflow models. DIF captures essential modeling information that is required in dataflow-based analysis and optimization techniques, such as algorithms for consistency analysis, scheduling, memory management, and block processing, while optionally hiding proprietary details such as the actual code that implements the dataflow blocks. Accompanying DIF is a software package of intermediate representations and algorithms that operate on application models that are captured through DIF. This paper describes the structure of the DIF language together with several implementation and usage examples.

1 Introduction

Modeling of DSP applications based on coarse-grain dataflow graphs is widespread in the DSP design community, and a large and growing set of DSP design tools support such dataflow semantics [2]. Since a variety of dataflow modeling styles and accompanying semantic constructs have been developed for DSP design tools (e.g., see [1, 4, 5, 8, 11, 14, 15]), a critical problem in the process of technology transfer to, from, and across such tools is a common, vendor-independent language, and associated suite of intermediate representations and algorithms for DSP-oriented dataflow modeling. This paper describes a preliminary version of a *dataflow interchange format* (*DIF*) for addressing this problem.

As motivated above, DIF is not centered around any particular form of dataflow, and is designed instead to express different kinds of dataflow semantics. Our present version of DIF includes built-in support for synchronous dataflow (SDF) semantics [14], which have emerged as an important common denominator across many DSP design tools and support powerful algorithms for analysis and software synthesis [3]. DIF also includes support for the closely related cyclo-static dataflow (CSDF) model [4], and has specialized support for various restricted versions of SDF, in particular, homogeneous and single-rate dataflow, which are often used in multiprocessor scheduling and hardware synthesis. Additionally, support for Boolean dataflow (BDF) [5] and parameterized dataflow [1], and for general constructs involving dynamic, variable-parameter dataflow quantities (production rates, consumption rates, and delays) is provided in DIF. DIF also captures hierarchy, and arbitrary non-dataflow attributes that can be associated with dataflow graph nodes (also called *actors* or *blocks*), edges, and graphs.

2 The Language

DIF is designed to be exported and imported automatically by tools. However, unlike other interchange formats, DIF is also designed to be read and written by designers who wish to understand the dataflow structure of applications or the dataflow semantics of a particular design tool, or who wish to specify an application model for one or more design tools using the features of DIF. *Indeed, DIF provides the programmer a unique, integrated set of semantic features that are relevant to dataflow modeling*. As a result, DIF is not based on XML, which is more for pure data exchange applications, and is not well-suited for being read or written by humans. Due to the emphasis on readability, DIF supports *C*/Java-style comments, allows specifications to be modularized across multiple files (through integration with the standard *C* preprocessor), and is based on a block-structured syntax.

A dataflow graph definition in DIF consists in general of six blocks of code: *topology*, *interface*, *refinement*, user-defined and built-in *attributes,* and *parameters*. These code blocks are contained in a main block defining the dataflow graph. Note that each block is optional without violating language basics. Using the *basedon* keyword, a graph can inherit the same topology as another graph while overriding arbitrary attributes and parameters. Figure 1 illustrates the general form of a graph definition block. The optional *keyword* on the first line denotes the type (form of dataflow). Further details on the different graph types available are described in Section 3.

2.1 Defining the Topology of a Dataflow Graph

The topology definition of a graph consists of node and edge definition blocks. These define the sets of nodes and edges, and associate a unique identifier with each node and each edge. Since dataflow graphs are directed graphs, edges are specified by their source and sink node identifiers. A node definition may also include a port association (described further in Section 2.2) for interfacing to other graphs. The lower left side of Figure 2 shows an example of a topology definition block.

2.2 Hierarchical Graphs

Given the importance of hierarchical design in graphical design tools, a necessary feature of the DIF language is the general ability to associate a node of a graph with a "nested" subgraph. Such hierarchical nodes are called *supernodes* in DIF terminology. In addition to providing for hierarchy, this supernode feature allows for reuse of graph specifications: a topological pattern that appears multiple times in a graph can be defined as a separate graph and every occurrence in the original graph (parent graph) or in multiple graphs can be replaced with a single node.

A graph can be declared as a nested subgraph in the *refinement* block of a parent graph. For a graph to be declared as a subgraph, it should have an *interface* block, which includes a list of directed ports. A port will then be associated either with a node (in the *topology* block) or with one of the ports of a super node (in the *refinement* block).

Further details and examples of the hierarchy mechanism in DIF can be found in [10].

```
[keyword] graph graphID [basedon graphID] {
   params {
      param prm1, prm2, ...;
      domain (prm1, {1, 2, ...});
      domain (prm2, [1, 5]);
      ...
   }
   interface {
      input portID portID ...;
      output portID portID ...;
   }
   topology {
      nodes { nodeID [:portID ] nodeID [:portID ] ...}
      edges {
         edgeID sourceINodeID sinkNodeID;
         edgeID sourceINodeID sinkNodeID;
         ...
      }
   }
   refinement {
      subgraphID nodeID
         subPortID:edgeID subPortID:PortID ...;
      subgraphID nodeID
         subPortID:portID subPortID:edgeID ...;
      ...
   }
   attribute attributeName {
      edgeID value ;
      nodeID value ;
      ...
   }
   ...
   [built-in attribute] {...}
   ...
}
```

Figure 1. A sketch of a dataflow graph definition in DIF. Items in boldface are DIF keywords. Italicized words are to be defined by the user. Parts in braces are optional.

2.3 User-defined and Built-in Attributes

DIF supports assigning attributes to nodes, edges, and graphs. There are two types of attributes: *user-defined* and *built-in*. *User-defined attributes* are attributes with arbitrary names that can take on any value assigned by the user. *Built-in attributes* are predefined attributes, which have associated keywords in the DIF language, and are usually handled in a special way by the compiler. An example of a built-in attribute is the *delay* parameter of graph edges.

2.4 Parameters

Parameterization of attribute values is possible in DIF with the *params* block. The capability of defining a possible set of values (*domain*) for an attribute instead of a specific value provides useful support for dynamic and reconfigurable dataflow graphs. The domain of a parameter can be an enumerated set of values, an interval, or a composition of both forms.

2.5 The *basedon* Feature

Using the *basedon* keyword, a graph that has the same topology as another graph, but with different attribute or parameter values can be defined concisely with just a reference to the other graph. The user can change selected parameter and attribute values by overriding them in *attribute* and *params* blocks of the new graph.

3 Dataflow Support

This *DIF package* is a Java-based software package for DIF that is being developed, along with the DIF language, at the University of Maryland. Associated with each of the supported dataflow graph types is an intermediate representation within the DIF package that provides an extensible set of data structures and algorithms for analyzing, manipulating, and optimizing DIF representations. Also, conversion algorithms between compatible graph types (such as CSDF to SDF or SDF to single-rate conversion) are provided. Presently, the collection of dataflow graph algorithms is based primarily on well-known algorithms (e.g., algorithms for iteration period computation [9], consistency validation [14], and loop scheduling [3]), and the contribution of DIF in this regard is to provide a common repository and front-end through which different DSP tools can have efficient access to these algorithms. We are actively extending this repository with additional dataflow modeling features and additional algorithms, including more experimental algorithms for data partitioning and hardware synthesis. Below is a summary of the dataflow models that are currently supported in DIF.

3.1 DIF Graphs

DIF graphs are the default and most general class of dataflow graphs supported by DIF. DIF graphs can be specified explicitly using the *dif* keyword. In DIF graphs, no restriction is made on the rate at which data is produced and consumed on dataflow edges, and other types of specialized assumptions, such as statically-known delay attributes, are avoided as well. In the underlying intermediate representation, an arbitrary Java object can be attached to each node/edge incidence to represent the associated dataflow properties. In the inheritance hierarchy of the DIF intermediate representations, DIF graphs are the base class of all other forms of dataflow. In this sense, all dataflow graphs modeled in DIF are instances of DIF graph. Furthermore, if a tool cannot export to any of the more specialized versions of dataflow supported by DIF, it should export to DIF graphs.

3.2 CSDF Graphs

In restricted versions of the DIF graph model that are recognized in DIF, the number of data values (tokens) produced and consumed by each node may be known statically and edge delays may be fixed integers. For example, *CSDF* graphs, based on the cyclo-static dataflow model [4], are specified by annotating DIF graph definitions with the *csdf* keyword. In CSDF graphs, production and consumption rates can vary between node executions, as long as the variation forms a certain type of periodic pattern. Consequently, values of these rates are integer vectors. These vectors are associated with CSDF graph edges using the *production* and *consumption keywords.* For example, the code fragment

```
production {e1 [1 1 2 4]; e2 [2 2 3];}
```

associates the periodic production patterns

$$(1, 1, 2, 4, 1, 1, 2, 4, \ldots) \text{ and } (2, 2, 3, 2, 2, 3, \ldots)$$

with edges $e1$, and $e2$, respectively.

3.3 SDF Graphs

Similar to CSDF graphs, token production and consumption rates of synchronous dataflow (SDF) graphs [14] are known at compile time, but they are fixed rather than periodic integer values. SDF graphs are specified using the *sdf* keyword, and the arguments of *production* and *consumption* specifiers in SDF graphs are required to be integers, as in:

```
production {e1 4; e2 3;}
consumption {e1 5; e2 2;}
delay {e1 1; e2 2;}
```

The last statement, which is permissible in other DIF graph types as well, associates integer-valued delays to the specified edges.

3.4 Single Rate and HSDF Graphs

Single rate graphs are a special case of SDF graphs where the production and consumption values on each edge are identical. In single rate graphs, nodes execute ("fire") at the same average rate [3]. In the slightly more restricted case of *homogeneous* SDF (HSDF) graphs, production and consumption values are equal to one for all edges. Instead of *production* and *consumption* attributes, DIF uses the *transfer* keyword for edges in single rate graphs. DIF does not associate an attribute for token transfer volume in HSDF since it is not variable.

3.5 Parameterized Dataflow Graphs

Parameterized dataflow [1] graphs can be represented in DIF using the parameterization and hierarchy facilities of DIF. Specifically, separate subgraphs can be defined for the *init*, *subinit*, and *body* subsystems of a parameterized dataflow model, and variable parameters with associated parameter value domains can be defined and linked to outputs of the init or subinit graphs through user-defined attributes.

3.6 Other Dataflow Graphs

Boolean-controlled dataflow (BDF) [6] is a form of dynamic dataflow for supporting data-dependent DSP computations. A dynamic actor produces or consumes a number of tokens depending on the incoming data values during each firing. In BDF, the number of tokens produced or consumed by a dynamic actor is restricted to be a two-valued function of the value of a control token. For example, the *Switch* actor in BDF consumes an input token and a control token. If the control token is true, the input token is sent to an outgoing edge labeled *True*, otherwise it is sent to an outgoing edge labeled *False*. BDF graphs are specified using the *bdf* keyword and a syntax is provided for specifying control inputs to BDF actors and their relationships to other incident edges.

Interval-Rate Locally-static Dataflow (ILDF) [18] is proposed to analyze dataflow graphs whose component data rates are not known precisely at compile time. In ILDF graphs, the production and consumption rates remain constant throughout execution (locally-static), but only the minimum and maximum values (interval-rate) of these constants are given. DIF is capable of representing ILDF graphs by parameterizing the production and consumption rates of ILDF edges and specifying the intervals of those parameters.

In addition to the aforementioned dataflow models, a variety of other dataflow models are being explored for inclusion in DIF.

4 DIF Language Implementation

The DIF package includes a parser that converts a DIF specification into a suitable, graph-theoretic intermediate representation based on the particular form of dataflow used in the DIF specification. This parser is implemented using a Java-based compiler-compiler called *SableCC* [7]. The flexible structure of the compiler enables easy extensibility for different graph types.

Using *DIF writer* classes, it is also possible to generate DIF files from intermediate representations (graph objects) in the DIF package. The default writer is the DIF graph writer, which generates a DIF graph specification, and custom writers can be constructed by extending the DIF graph writer base class to handle semantic additions/restrictions by converting them to appropriate built-in attributes, structural conventions, etc.

The DIF package builds on some of the packages of Ptolemy II [13]. In particular, the attribute features of DIF are built on the rich classes for managing attributes in Ptolemy II, and the intermediate representations of DIF build on the *graph* package of

Ptolemy II, which provides data structures and algorithms for working with generic graphs.

5 EXAMPLES

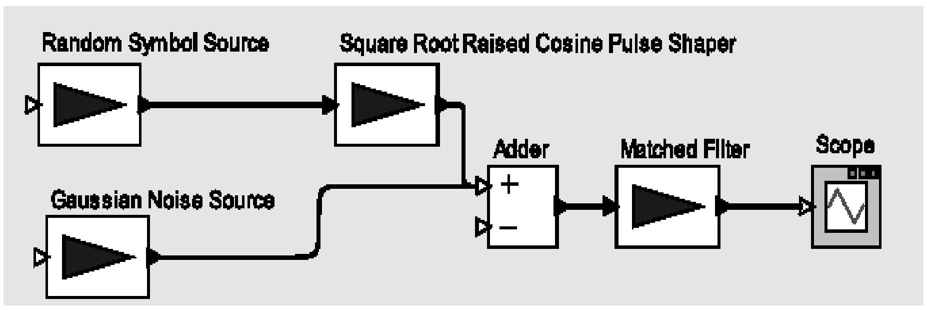
This section illustrates some further examples of the utility of the DIF package.

5.1 Ptolemy

We have developed a back-end for Ptolemy II that generates DIF graphs from dataflow-based Ptolemy II models. An example of Ptolemy-to-DIF conversion through this back-end is shown in Figure 2. A front-end that converts DIF specifications into Ptolemy II models is under development.

5.2 MCCI Autocoding Toolset

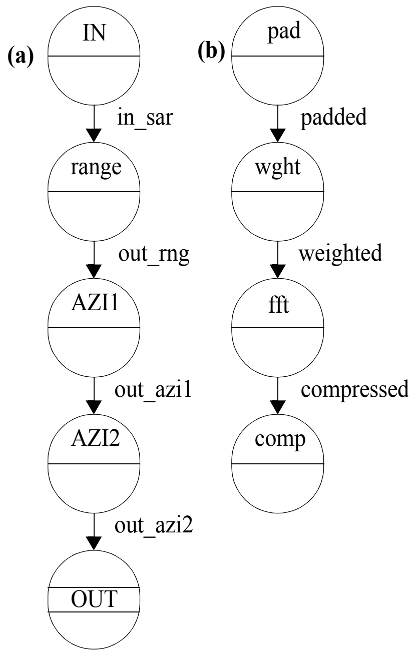
Another usage example of DIF is in the Autocoding Toolset of Management, Communications, and Control Inc. (MCCI) [16]. This tool is designed for mapping large, complex signal processing applications onto high-performance multiprocessor platforms.



```
sdf graph _graph {
    topology {
        nodes {n0 n1 n2 n3 n4 n5}
        edges {e0 n0 n1;
            e1 n1 n2; e2 n2 n4;
            e3 n3 n2; e4 n4 n5;}}
    production {e0 1; e1 16;
        e2 1; e3 1; e4 1;}
    consumption {e0 1; e1 1; e2 1;
        e3 1; e4 1;}
    delay {e0 0; e1 0; e2 0; e3 0;
        e4 0;}
    computation {
        n0 DiscreteRandomSource;
        n1 RaisedCosine;
        n2 AddSubtract;
        n3 Gaussian;
        n4 RaisedCosine;
        n5 SequenceScope;}
}
```

Figure 2. Ptolemy II model of a PAM communication system that is exported to DIF. This example represents the functionality of each node as a *computation* attribute, which is derived from the Ptolemy II library definition.

Through a DIF-generating back-end developed at MCCI, the Autocoding Toolset supports generation of DIF specifications after partitioning the application.



(c)

```
graph rangeGraph {
   interface {input rng_in; output rng_out;}
   topology {
      nodes {pad:rng_in wght fft comp:rng_out}
      edges {padded pad wght; weighted wght fft; compressed fft comp;}}
   production {padded 1048576; weighted 1048576; compressed 1048576;}
   consumption {padded 1048576; weighted 1048576; compressed 1048576;}
   delay {padded 0; weighted 0; compressed 0;}
}
```

(d)

```
graph SAR {
   ...
   refinement {
      rangeGraph range rng_in:in_sar rng_out:out_rng;
   }
   ...
}
```

Figure 3. (a) The top-level partitioned application graph of a SAR application in the MCCI Autocoding Toolset. (b) Range processing. (c) Range processing in DIF. (d) Range processing instantiation in SAR. Note that although 3(c) is a single rate graph, the Autocoding Toolset presently exports this in the more general form of a DIF graph. This example is adapted due to space constraints.

Figure 3 shows a synthetic aperture radar (SAR) application developed in the Autocoding Toolset. The functional requirements of SAR processing consist of four logical processes: data input and conditioning, range processing, azimuth processing and data output. The Autocoding Toolset partitions the application into five parts dividing the azimuth processing into two parts. Figure 3(a) shows the top level functional definition graph and Figure 3(b) shows the *range* subgraph with its DIF definition. *Range* processing of data includes conversion to complex floating point numbers, padding the end of each data row with zeros, multiplying by a weighting function, computing the FFT, and multiplying the data by the radar cross-section compensation.

5.3 Visualization and Benchmark Generation

The DIF package contains facilities to generate DIF specifications of randomly-generated, synthetic benchmarks. This can be useful for more extensive testing of tools and algorithms beyond the set of available application models. The benchmark generator is based on an implementation of Sih's dataflow graph generation algorithm [17], which constructs application-like graphs by mimicking patterns found in practical dataflow models.

DIF specifications and intermediate representations can also be converted automatically into the input format of *dot* [12], a well-known graph-visualization tool.

6 Summary

This paper has presented the dataflow interchange format (DIF), a textual language for writing coarse-grain, dataflow-based models of DSP applications, and for communicating such models between DSP design tools. The objectives of DIF are to accommodate a variety of dataflow-related modeling constructs, and to facilitate experimentation with and technology transfer involving such constructs. We are actively extending the DIF language, including the set of supported dataflow modeling semantics, and the associated repository of intermediate representations and algorithms.

7 Acknowledgements

This research is sponsored in part by DARPA (contract #F30602-01-C-0171, through the USC Information Sciences Institute), and the Semiconductor Research Corporation (contract #2001-HJ-905).

References

1. B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408-2421, October 2001.

2. S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for DSP. *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, 47(9):849-875, September 2000.
3. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 21(2):151-166, June 1999.
4. G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *Proc. ICASSP*, pages 3255-3258, May 1995.
5. J. T. Buck and E. A. Lee. Scheduling dynamic dataflow graphs using the token flow model. In *Proc. ICASSP,* April 1993.
6. J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Tech. Report UCB/ERL 93/69, Ph.D. Thesis, Dept. of EECS, University of California, Berkeley, 1993.
7. E. Gagnon. *SableCC, an object-oriented compiler framework*. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, March 1998.
8. G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved programs for DSP computation. In *Proc. ICASSP*, March 1992.
9. K. Ito and K. K. Parhi. Determining the iteration bounds of single-rate and multi-rate dataflow graphs. In *Proc. IEEE Asia-Pacific Conference on Circuits and Systems*, December 1994.
10. F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya. First version of a dataflow interchange format. Technical report, Institute for Advanced Computer Studies, University of Maryland at College Park, November 2002.
11. B. Kienhuis and E. F. Deprettere. Modeling stream-based applications using the SBF model of computation. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 385-394, September 2001.
12. E. Koutsofios and S. C. North. *dot* user's manual. Technical report, AT&T Bell Laboratories, November 1996.
13. E. A. Lee. Overview of the Ptolemy project. Technical Report UCB/ERL M01/11, Department of EECS, UC Berkeley, March 2001.
14. E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235-1245, September 1987.
15. M. Pankert, O. Mauss, S. Ritz, and H. Meyr. Dynamic data flow and control flow in high level DSP code synthesis. In *Proc. ICASSP*, 1994.
16. C. B. Robbins. *Autocoding Toolset software tools for automatic generation of parallel application software*. Technical report, Management, Communications & Control, Inc., 2002.
17. G. C. Sih. *Multiprocessor Scheduling to account for Interprocessor Communication*. Ph.D. thesis, Department of EECS, UC Berkeley, April 1991.
18. J. Teich and S. S. Bhattacharyya. Analysis of dataflow programs with interval-limited datarates. In *Proceedings of the International Workshop Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2004. To appear.