Technical Report UMIACS-TR-2007-32,
Institute for Advanced Computer Studies,
University of Maryland at College Park, June 2007

Dataflow Interchange Format: Language Reference for DIF Language Version 1.0 User's Guide for DIF Package Version 1.0

Chia-Jui Hsu, Ivan Corretjer, Ming-Yung Ko, William Plishker, Shuvra S. Bhattacharyya

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

June 16, 2007

1 Introduction

1.1 What is the Dataflow Interchange Format?

The dataflow interchange format (DIF) is a standard language for specifying mixed-grain dataflow models for digital signal processing (DSP) systems and other streaming-related application domains. Here, by *DSP*, we mean applications that processes digitally represented signals, including signals associated with audio, video, image, digital communications, and multimedia data. We say that DIF is a *standard* language, because it is designed with the primary goal of providing a unifying framework for representing dataflow graphs that may arise in a wide variety of DSP applications, and a wide variety of more specialized DSP design tools. Other key objectives of the DIF project are to provide an extensible repository for representing, experimenting with, and developing dataflow model of computations, and associated analysis techniques, to facilitate technology transfer of applications across different DSP design tools, and different embedded processing platforms. This report introduces the DIF language, the DIF package (a software package that accompanies the DIF language), the dataflow models that are supported in DIF, the approach to exporting and importing DIF representations, and a methodology that is supported by DIF for porting DSP applications across different DSP tools and platforms.

1.2 Why use the Dataflow Interchange Format?

Modeling DSP applications through coarse-grain dataflow graphs is widespread in the DSP design community, and a variety of dataflow models have been developed for dataflow-based design (e.g., see [2, 5, 7, 8, 16, 22]). A growing set of DSP design tools support such dataflow semantics [4]. Furthermore, Turing-complete DSP-oriented dataflow modeling approaches are available to provide for full expressibility within the dataflow framework (see, for example Section 4.4).

A critical issue arises in transferring technology across these design tools due to the lack of a standard and vendor-independent language and an associated package with intermediate representations and efficient implementations of dataflow analysis and optimization algorithms. DIF is designed for this purpose and is proposed to be a standard language for specifying and working with dataflow-based DSP applications across all relevant dataflow modeling approaches that are related to DSP system design.

In order to provide the DSP design industry with a convenient front-end to use DIF and the DIF package, automating the exporting and importing processes between DIF and design tools is an essential feature. Although problems related to exporting and importing are design-tool-specific, many practical implementation issues are quite common among different design tools. DIF and the associated DIF package have been designed to help reuse effort that is related to these common issues so that developers and users of design tools can focus on the novel features and unique constraints associated with their design problems.

As a related point, the problem of transferring DSP applications across design tools with a high degree of automation has also been considered throughout the development of DIF. Such porting typically requires tedious effort, is highly error-prone, and is very fragile with respect to changes in the application model being ported (changes to the model require further manual effort to propagate to the ported version). This motivates a new approach to porting DSP applications across dataflow-based design tools through the interchange information captured by the DIF language, and through additional infrastructure and utilities to aid in conversion of complete data-

flow-based application models (including all dataflow- and actor-specific details) to and from DIF.

Portability of DSP applications across design tools is equivalent to portability across all of the underlying embedded processing platforms and DSP code libraries supported by those tools. Such portability would clearly be a powerful capability if it can be attained through a high degree of automation, and a correspondingly low level of manual or otherwise ad-hoc fine-tuning. The key advantage of using a DIF specification as an intermediate state in achieving such efficient porting of DSP applications is the comprehensive representation in the DIF language of functional semantics and component/subsystem properties that are relevant to design and implementation of DSP applications using dataflow graphs.

1.3 DIF-based Design Methodology

DIF is not just a language format, but the foundation for a productive, efficient design flow for DSP applications. Our overall end-user vision of DIF is to use it in the context of the design methodology outlined in Figure 1. Application designers begin by developing an application in a specific semantic framework with a set of library elements to aid in the description of the application. Designers may describe their applications directly in DIF or develop a translator that generates DIF code automatically. DSP applications specified by (or translated into) the DIF language are referred to as *DIF specifications*. The DIF package includes a frond-end tool, the DIF language parser, which converts a DIF specification into a corresponding graph-theoretic intermediate representation. This parser is implemented using a Java-based compiler-compiler called SableCC [12].

For supported dataflow models, the DIF package also provides efficient implementations of various utilities for working with dataflow graphs that operate on DIF intermediate representations. These implementations provide designers a convenient interface for analyzing and optimizing DSP applications.

The DIF package provides an intermediate layer between abstract dataflow models and different practical implementations. DIF exporting and importing tools automate the process of translating between tool-specific dataflow graph formats and DIF specifications, and provide a useful front-end for using DIF and the DIF package. Dataflow-based DSP design tools that we have been experimenting with in our development of DIF so far are the Agilent Technologies ADS environment [18]; CAL actor language [10]; National Instruments LabVIEW [1]; Management Communications and Control, Inc. (MCCI) Autocoding Toolset [19]; and synchronous dataflow domain of Ptolemy II [11]. However, DIF is in no way designed to be specific to these tools; these tools are used only as a starting point for experimenting with DIF in conjunction with sophisticated academic and industrial DSP design tools. Tools such as these form a layer in our proposed DIF-based design methodology. The *embedded processing platforms* layer in Figure 1 gives examples of platforms supported by Ptolemy II and the Autocoding Toolset. In general, this layer represents all embedded platforms that are supported by dataflow-based DSP design tools.

1.4 Evolution of DIF

DIF's foundation is in academic research and this foundation has been described in the literature. In the first version of DIF [13, 14], we demonstrated the capability of conveniently specifying and manipulating fundamental dataflow models, such as synchronous dataflow [16] and cyclo-static dataflow [8]. Nonetheless, its semantics were insufficient to describe in detail more

advanced dataflow semantics, and to specify actor-specific information. As a result, the DIF language was further developed to the second version, version 0.2, for supporting an additional set of important dataflow models of computation and facilitating design-tool-dependent transferring processes. We have shown the feasibility of the DIF porting capabilities by demonstrating the porting of a synthetic aperture radar application from the MCCI Autocoding Toolset [19] to Ptolemy II [11].

Note that any dataflow semantics can be specified using the "DIF" model of dataflow supported by DIF and the corresponding DIFGraph intermediate representation. However, for performing sophisticated analyses and optimizations for a particular dataflow model of computation, it is often useful to have more detailed and customized features in DIF that support the model. This is why the exploration of different dataflow models for incorporation into DIF is an ongoing area for further development of the language and software infrastructure.

From version 0.1 to version 0.2, the syntax consistency and code reusability support of DIF have been improved significantly. DIF language version 0.2 also supports more flexible parameter assignment and provide more flexible treatment of graph attributes. Moreover, it supports most commonly used value types in DSP applications and provides arbitrary naming spaces. Also, per-

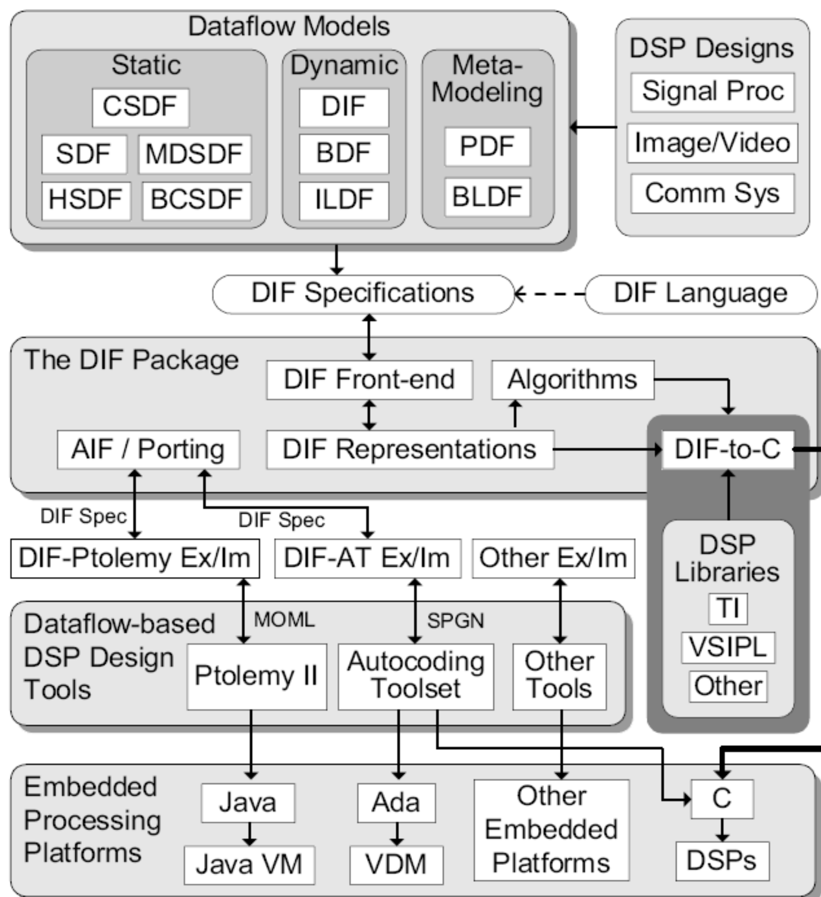


Figure 1. DIF-based design flow.

haps most significantly, the *actor* block is newly created in DIF version 0.2 for specifying design-tool-dependent actor information.

1.5 Updates to this Document

Revisions to this document may be made to incorporate minor fixes and clarifications as we become aware of the need for them. Such revisions will be posted at

`http://www.ece.umd.edu/DSPCAD/dif/`

Information about the specific version of this document can be found in Section 10.

2 Dataflow Graphs and Hierarchical Dataflow Representation

Theory based on dataflow graphs form the foundation of the Dataflow Interchange Format. The following section describes formally what a dataflow graph is and what semantics are implicit to it.

2.1 Dataflow Graphs

In the dataflow modeling paradigm, computational behavior is depicted as a dataflow graph (DFG). A dataflow graph G is an ordered pair (V,E), where V is a set of vertices, and E is a set of directed edges. A directed edge $e = (v1,v2) \in E$ is an ordered pair of a source vertex $src(e)$ and a sink vertex $snk(e)$, where $src(e) \in V$, $snk(e) \in V$. An edge e can be denoted as $v1 \rightarrow v2$, where $v1$ and $v2$ are, respectively, the source and sink vertices. Given a directed graph $G = (V, E)$ and a vertex $v \in V$, the set of incoming edges of v is denoted as $in(v) = \{e \in E | snk(e) = v\}$, and similarly the set of outgoing edges of v is denoted as $out(v) = \{e \in E | src(e) = v\}$.

In dataflow graphs, a vertex v (also called a *node*) represents a computation and is often associated with a node *weight*. The weight of an object in DIF terminology refers to arbitrary information that a user wishes to associate with the object (e.g., the execution time of a node or the type of data transferred along an edge). An edge e in dataflow graphs is a logical data path from its source node to its sink node. It represents a FIFO (first-in-first-out) queue that buffers data values (tokens) that are destined for its sink node. An edge has a non-negative integer delay $delay(e)$ associated with it and each delay unit is functionally equivalent to the z^{-1} operator in signal processing.

Dataflow graphs naturally capture the data-driven property that is inherent in many DSP computations. An actor (node) can fire (execute) at any time when it is *enabled* (the actor has sufficient tokens on all its incoming edges to perform a meaningful computation). When firing, it consumes certain numbers of tokens from its incoming edges $in(v)$, executes the computation, and produces certain numbers of tokens on its outgoing edges $out(v)$. This combination of consumption, execution, and production may or may not be carried out in an interleaved manner. Given an edge $e = (v1,v2)$ in a dataflow graph, if the number of tokens produced on e by an invocation of $v1$ is constant throughout execution of the graph, then this constant number of tokens produced is called the *production rate* of e and is denoted by $prd(e)$. The *consumption rate* of e is defined in an analogous fashion, and this rate, when it exists, is denoted by $cns(e)$.

2.2 Hierarchical Structure

In dataflow-based DSP systems, the granularity of actors can range from elementary operations such as addition, multiplication, or logical operations to DSP subsystems such as filters or fast Fourier transform computations. If an actor represents an indivisible operation in some context, it is called *atomic*. An actor that represents a hierarchically-nested subgraph is called a *supernode;* an actor that does not is called a *primitive node*. The *granularity* of a dataflow actor describes its functional complexity. Simple primitive actors such as actors for addition or for elementary logical operations are called *fine-grained* actors. If the actor complexity is at the level of signal processing sub-tasks, the actor is called *coarse-grained.* Practical dataflow models of applications typically contain both fine- and coarse-grained actors; dataflow graphs underlying such models are called *mixed-grain* graphs.

In a dataflow representation for a sophisticated DSP application, the mixed-grain dataflow representation of the overall system may consist of several supernodes, where each top-level supernode can be further refined into another mixed-grain dataflow graph, possibly with additional (nested) supernodes.

One way to describe such a complicated system is to flatten the associated hierarchy into a single non-hierarchical graph that contains no supernodes. However, such an approach may not always be useful for the following reasons. First, analyzing a dataflow graph with the original hierarchical information intact may be more efficient than trying to analyze an equivalent flattened graph that is possibly much larger. Second, the top-down design methodology is highly applicable to DSP system design, so the overall application is usually most naturally represented as a hierarchical structure. Thus, incorporating hierarchy information into the DIF language and graph representations is an essential consideration in the DIF project.

Definitions related to hierarchies are introduced as follows. A *supernode* s in a graph $G = (V, E)$ represents a dataflow subgraph G', and this association is denoted as $s \cong G'$. The collection of all supernodes in G forms a subset S in V such that $s \in S \subset V$ and $\forall v \in \{V - S\}$, v is a primitive node. If a supernode s in G represents the nested graph G', then G' is called a *subgraph* of G and G is the *supergraph* of G'.

A *hierarchy* $H = (G, I, M)$ contains a graph G with an interface I, and a mapping M. Given another hierarchy $H' = (G', I', M')$, if G' is a subgraph of G, we said that H' is a sub-hierarchy of H and H is a super-hierarchy of H'.

A *mapping* from a supernode s representing subgraph G' to a sub-hierarchy H' containing G' is denoted as $s \Rightarrow H'$, where $H' = (G', I', M')$. In such a case, we write that $s \cong G'$, and $hierarchy(s) = H'$. The *mapping* M in a hierarchy $H = (G, I, M)$ is the set that contains all mappings (to subhierarchies) of supernodes s in $G = (V, E)$; that is,

$$M = \{s \Rightarrow hierarchy(s) | (s \in S)\},$$

where S is the set of supernodes in G.

The *interface* I in a hierarchy H is a set consisting of all interface ports in H. An *interface port* (or simply *port*) p is a dataflow gateway through which data values (tokens) flow into a graph or flow out of a graph. From the interior point of view, a port p can associate with one and only one node v in graph G, and this association is denoted as $p{:}v$, where $p \in I$, $v \in V$, $G = (V, E)$ and $H = (G, I, M)$. From the exterior point of view, a port p can either connect to one and only one edge e'' in graph G'' or connect to one and only one port p'' in hierarchy H'', where G'' is the supergraph of G, and H'' is a super-hierarchy of H. These connections are denoted as $p \sim e''$ and $p \sim p''$ respectively, where $p \in I$, $e'' \in E''$, $p'' \in I''$, $H = (G, I, M)$, $G'' = (V'', E'')$, and $H'' = (G'', I'', M'')$.

An interface port is directional; it can either be an *input port* or an *output port*. An input port is an entry point for tokens flowing from outside the hierarchy to an interior node, and conversely, an output port is an exit point for tokens moving from an interior node to somewhere outside the hierarchy. Given $H = (G, I, M)$, $in(I)$ denotes the set of input ports of H and $out(I)$ denotes the set of output ports of H, where $in(I) \cap out(I) = \varnothing$, and $in(I) \cup out(I) = I$. Then given a port $p \in in(I)$, $p{:}v$, and $p \sim e''$, v consumes tokens from e'' when firing. Similarly, given $p \in out(I)$, $p{:}v$, and $p \sim e''$, v produces tokens to e'' when firing.

The association of an interface port with an interior node and the connection of an outer edge to an interface port can facilitate the clustering and flattening processes. For example, given $p{:}v$, $p \sim e''$, $src(e'') = v''$, $snk(e'') = s$, $s \Rightarrow H = (G, I, M)$, and $p \in I$, a new edge e can be connected from v'' to v directly after flattening the hierarchy H.

With the formal dataflow graph definition reviewed in Section 2.1 and the hierarchical structures defined in this section, we are able to precisely discuss how hierarchical dataflow graphs are represented in the DIF language. The DIF language introduced in detail in the following section.

3 The DIF Language

The Dataflow Interchange Format (DIF) is proposed to be a standard language for specifying dataflow semantics in dataflow-based application models for DSP system design. This language is suitable as an interchange format for different dataflow-based DSP design tools because it provides an integrated set of syntactic and semantic features that can fully capture essential modeling information of DSP applications without over-specification.

From the 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. Therefore, the dataflow semantics of a DSP application are unique in DIF regardless of any design tool used to originally enter the application specification. Moreover, DIF also provides syntax to specify design-tool-specific information, and such tool-specific information is captured within the data structures associated with the DIF intermediate representations. Although this information may be irrelevant to many dataflow-based analyses, it is essential in exporting, importing, and transferring across tools, as well as in code generation.

DIF is not aimed to directly describe detailed executable code. Such code should be placed in actual implementations, or in libraries that can optionally be associated with DIF specifications. Unlike other description languages or interchange formats, the DIF language is also designed to be read and written by designers who wish to specify DSP applications in dataflow graphs or understand applications based on dataflow models of computations. As a result, the language is clear, intuitive, and easy to learn and use for those who have familiarity with dataflow semantics.

A DIF specification is in general made up of four standard blocks — `basedon`, `topology`, `interface`, and `parameter`; zero or more `refinement` blocks; zero or more `actor` blocks; zero or more user-defined `attribute` blocks; and zero or more built-in attribute blocks. Thus, a total of eight kinds of blocks make up a DIF specification. These blocks specify different aspects of dataflow semantics and modeling information. The following subsections introduce the syntax of the DIF language. In some code segments, DIF keywords are highlighted in boldface for emphasis. Non-bold words are either keywords (when keyword highlighting is not used), or text items that need to be specified by users.

3.1 The Main Block

A dataflow graph is specified in the main (top-level) block along with two arguments — *dataflowModel* and the *graphID*. The *dataflowModel* keyword specifies the specific dataflow model of computation that underlies the graph, and the *graphID* specifies a name (identifier) for the graph. Figure 2 illustrates the overall structure of the main block. The open and close braces (`{` and `}`) are required in the DIF code here to delimit the various blocks that are involved in the specification.

The eight kinds of blocks that make up a DIF specification are defined within the main (top-level) pair of braces. For each block apart from those corresponding to user-defined attributes, the block starts with a block-related DIF keyword and the contents of the block are enclosed by braces. A block corresponding to a user-defined attribute starts with the keyword `attribute`, followed by a user-defined identifier, which gives a name for the user-defined attribute. User-defined attributes provide a flexible mechanism by which users can effectively extend the DIF language and experiment with the DIF framework according to different specialized needs. User-defined attributes are discussed further in Section 3.8.

Statements inside block braces end with semicolons. Conventionally, identifiers and keywords in DIF are case-sensitive, and only consist of alphabetic characters, the underscore ('_') character, and digit characters. However, DIF also supports arbitrarily-composed identifiers by enclosing such identifiers between pairs of dollar-sign ('$') characters. This is useful in certain interchange applications, for example when interfacing to a tool that uses a broader set of characters in its identifiers.

As mentioned above, eight kinds of blocks can be used to make DIF specification. All of these blocks are optional. When they are present, the `basedon`, `topology`, `interface`, `parameter` and `refinement` blocks should be defined in this particular order. For example, if only `basedon`, `topology`, and `refinement` blocks are used in a specification, then the specification should consist of a `basedon` block, a `topology` block, and a `refinement block`, in that specific order.

3.2 The Basedon Block

```
basedon { graphID; }
```

The *basedon* block provides a convenient way to refer to a pre-defined graph, which is specified by *graphID* in the code template above. As long as the referenced graph has compatible topology, interface, and refinement blocks, designers can simply refer to it and override the name, parameters and attributes to instantiate a new graph. In many DSP applications, a set of different subgraphs can have the same topology but different sets of parameters or attributes. The `basedon` block is designed to support this characteristic and promote conciseness and code reuse when working with such subgraphs.

3.3 The Topology Block

The *topology* block is illustrated in Figure 3. The `topology` block specifies the topology $G = (V, E)$ (vertices and edges) of a dataflow graph. It consists of a node definition statement defining every node $v \in V$ and an edge definition statement defining every edge $e = (v_i, v_j) \in E$.

The keyword `nodes` is the keyword that starts a node definition statement, and user-defined node identifiers, *nodeID*s, are listed following the `nodes` keyword and equal sign. Similarly, `edges` is the keyword that starts an edge definition. An edge definition

```
dataflowModel graphID {
   basedon { ... }
   topology { ... }
   interface { ... }
   parameter { ... }
   refinement { ... }
   builtInAttr { ... }
   attribute usrDefAttrID { ... }
   actor nodeID { ... }
}
```

Figure 2. The overall structure of a DIF specification.

`edgeID (sourceNodeID, sinkNodeID)`

consists of three arguments: the identifier *edgeID* to use when referring to the edge; the source node identifier *sourceNodeID*, which specifies the source node for the edge; and the sink node identifier *sinkNodeID*, which specifies the sink node for the edge.

If there is no `topology` block in a DIF specification, then this indicates an empty dataflow graph (a graph with no vertices nor edges). An empty dataflow graph is a valid abstraction, and can be useful, for example, in applications where dataflow graphs are constructed entirely at run-time — in such cases, the static DIF specification would serve just to initialize the graph.

3.4 The Interface Block

Figure 4 illustrates the `interface` block. The `interface` block defines the interface I of a hierarchy $H = (G, I, M)$. An input definition statement defines every input port $p_i \in in(I)$ and the corresponding interior association $p_i{:}v_i$. Similarly, an output definition statement defines every output port $p_o \in out(I)$ and the corresponding interior association $p_o{:}v_o$, where $v_i, v_o \in V$, and $G = (V, E)$.

The keywords `inputs` and `outputs` are the keywords for input and output definition statements. Following the `inputs` or `outputs` keyword, port definitions are listed. A port definition

`portID : assocNodeID`

involves in general two arguments, a port identifier and its associated node identifier. DIF permits defining an interface port without an associated node, so `assocNodeID` is optional.

3.5 The Parameter Block

In many DSP applications, designers often parameterize important attributes such as the frequency of a sine wave generator, and the order of a fast Fourier transform computation. In interval-rate, locally-static dataflow [23], unknown production and consumption rates are specified by their minimum and maximum values. In parameterized dataflow [2], production and consumption rates are even allowed to be unspecified and dynamically parameterized. The *parameter* block is

```
topology {
   nodes = nodeID, ..., nodeID;
   edges = edgeID (sourceNodeID, sinkNodeID),
           ...,
           edgeID (sourceNodeID, sinkNodeID);
}
```

Figure 3. Illustration of the `topology` block.

```
interface {
   inputs = portID : assocNodeID, ..., portID : assocNodeID;
   outputs = portID : assocNodeID, ..., portID : assocNodeID;
}
```

Figure 4. Illustration of the `interface` block.

designed to support parameterizing values in ways like these, and to generally support attribute value ranges, attributes whose values are unspecified.

Figure 5 illustrates the DIF `parameter` block. In a parameter definition statement, such as that illustrated by Figure 5, a parameter identifier `paramID` is defined and an associated value is optionally specified. DIF supports various value types; these types are introduced in Section 3.10.

DIF also supports specifying a range of possible values for a parameter. For this purpose a *range* can be specified as an interval such as `(1, 2)`, `(3.4, 5.6]`, `[7, 8.9)`, or `[-3.1E+3, +0.2e-2]`; a set of discrete numbers such as `{-2, 0.1, +3.6E-9, -6.9e+3}`; or a combination (union) of intervals and discrete sets such as

`(1, 2) + (3.4, 5.6] + [7, 8.9) + {-2, 0.1, +3.6E-9, -6.9e+3}`.

3.6 The Refinement Block

Figure 6 illustrates the `refinement` block. The `refinement` block is used to represent hierarchical graph structures. If S denotes the set of supernodes in a given dataflow graph $G = (V, E)$, then for each supernode $s \in S$ there should be a corresponding `refinement` block in the DIF specification for G to specify the supernode-subgraph association $s \Rightarrow H'$. In addition, for every port $p' \in I'$ in sub-hierarchy $H' = (G', I', M')$, the connection $p' \sim e$, or $p' \sim p$ is also specified in this `refinement` block, where $e \in E$, $p \in I$, $H = (G, I, M)$, and H is the super-hierarchy of H'. Moreover, the unspecified parameters (parameters whose values are unspecified, e.g., because they may be unknown in advance, computed at runtime, or both) in subgraph G' can also be specified by parameters in G.

Each `refinement` block consists of three types of definitions. First, a subgraph-supernode refinement definition, `subgraphID = supernodeID`, defines $s \cong G'$. Second, subgraph interface connection definitions, `subportID : edgeID` or `subportID : portID`, describe $p' \sim e$ or $p' \sim p$. Third, a subgraph parameter specification, `subParamID = paramID`, specifies values for blank parameters in the subgraph by using parameters defined in the enclosing graph.

```
parameter {
   paramID = value;
   paramID : range;
   paramID;
   ...,
}
```

Figure 5. Illustration of the `parameter` block.

```
refinement {
   subgraphID = supernodeID;
   subportID : edgeID;
   subportID : portID;
   subParamID = paramID;
   ...;
}
```

Figure 6. Illustration of the `refinement` block.

Figure 7 illustrates how to use DIF to specify hierarchical dataflow graphs. In Figure 7, there are two dataflow graphs, *G1* and *G2*, and supernode *n6* in graph *G2* represents the subgraph *G1*. The corresponding DIF specification is also presented in Figure 7.

3.7 Blocks for Built-in Attributes

A number of built-in attributes for annotating dataflow modeling components are provided in DIF. For example the production and consumption rates of synchronous dataflow edges are supported as DIF built-in attributes. For each built-in attribute, there is a corresponding keyword in DIF that is used to specify built-in attribute blocks in DIF code associated with the built-in attribute. These blocks are used to specify values for the associated attribute for different modeling components that are associated with the attribute.

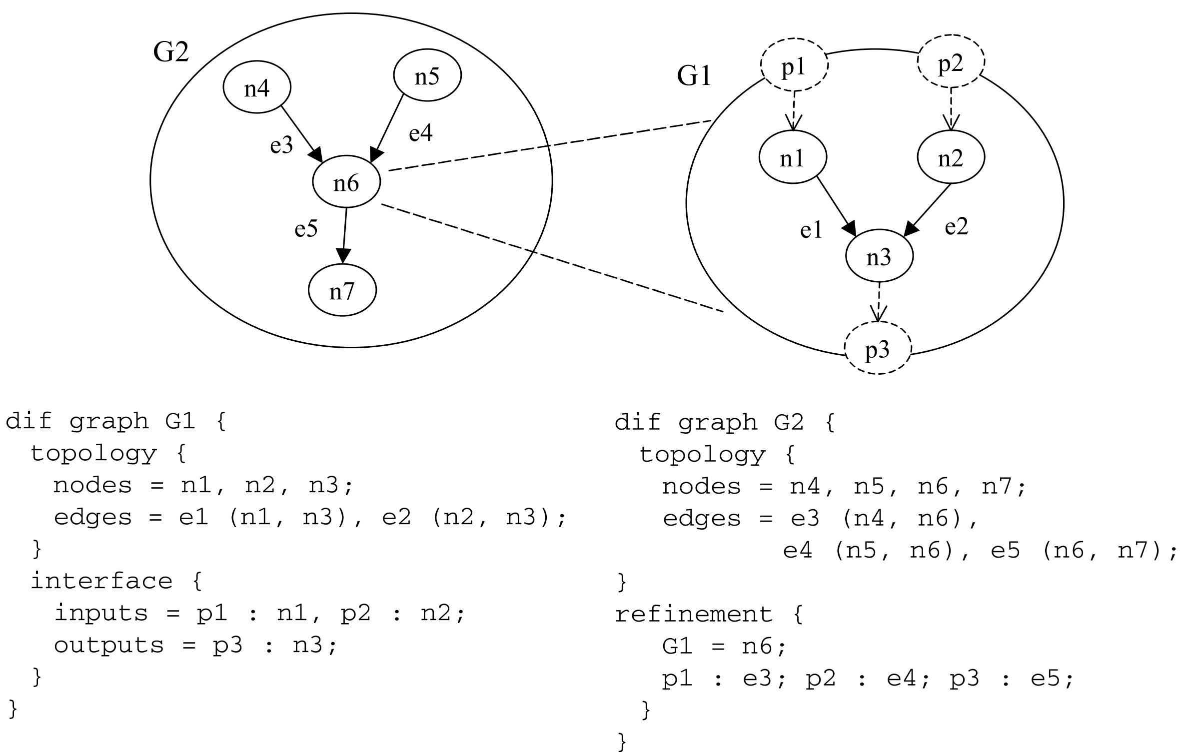


Figure 7. Hierarchical graphs and the corresponding DIF specifications.

```
builtInAttrID {
   elementID = value;
   elementID = ID;
   elementID = ID1, ID2, ..., IDn;
}
```

Figure 8. Illustration of a built-in attribute block.

Figure 8 illustrates a built-in attribute block. Here, *builtInAttrID* represents a place holder for one of the DIF keywords that are associated with supported built-in attributes. This specified built-in attribute keyword points out which built-in attribute the block corresponds to. In each of the *attribute value assignments* that make up the block (see Figure 8), the element identifier *elementID* can be a node identifier, an edge identifier, or a port identifier that specifies a specific modeling component that uses the built-in attribute. Multiple modeling components can use a given built-in attribute. In a given attribute value assignment, the *elementID* specifier (and the following equals sign) can be omitted; in this case, the enclosing graph uses the built-in attribute, and the value assignment specifies the value of the attribute as it applies to the graph.

DIF supports assigning attributes by using a 1) a value from a variety of possible value types (see Section 3.10 for a discussion of the supported value types); 2) an identifier; or 3) a list of identifiers. These forms are all illustrated in Figure 8. Any combination of these different forms can be used in a given built-in attribute block.

Note that some built-in attributes pertain only to specific dataflow models (e.g. synchronous dataflow) or to specific subsets of dataflow models. It is generally a syntax error for such built-in attributes to use in other contexts.

Three built-in attributes — `production`, `consumption`, and `delay` — are associated with edges in a variety of dataflow models that are supported by DIF. For example, if *e1* and *e2* have 1 and 2 units of delay, respectively, then a `delay` attribute block for these two edges can be specified as shown in Figure 9.

Note that the built-in attributes `production` and `consumption` are not exclusive to edges. In hierarchically nested dataflow models, a node associated with an interface does not have an edge associated with the interface connection. In such cases, specifying `production` and `consumption` values as port attributes is permitted in DIF.

3.8 Blocks for User-Defined Attributes

User-defined attribute blocks allow users to define and specify values for user-specific attributes. This is useful in adding specialized annotations to a DIF representation for use in non-

```
delay {
   e1 = 1;
   e2 = 2;
}
```

Figure 9. An example of a `delay` attribute block.

```
attribute usrDefAttrID {
   elementID = value;
   elementID = ID;
   elementID = ID1, ID2, ..., IDn;
}
```

Figure 10. Illustration of a user-defined attribute block.

standard or experimental contexts. The syntax is the same as for built-in attribute blocks, except that a user-defined attribute block starts with the `attribute` keyword followed by the associated user-defined attribute identifier *usrDefAttrID*. The format of a user-defined attribute block is illustrated in Figure 10.

3.9 Actor Blocks

The keyword `actor` is used to specify an *actor block*. An actor block is used to specify actor-specific information that complements the graph level information that is emphasized in other parts of a DIF specification. An actor block involves a set of attribute value assignments for *actor attributes*, and thus the structure of an actor block is somewhat similar to that of blocks for built-in and user-defined attributes. Additionally, a set of *built-in actor attributes* is provided to support certain commonly-used actor annotations that are suitable for incorporation in actor blocks. In the present version of DIF, this set only contains one member — the `computation` attribute, which is described below.

The format of an actor block is illustrated in Figure 11. Here, `computation` is a built-in actor attribute that is used to specify in some way the funcationality that is associated with an actor. For example, the value of this attribute could be the name of a library module that contains code to implement the actor. Other actor information within an actor block is specified through *user-defined actor attributes*. The lines in Figure 11 that start with *attributeID* illustrate the different forms in which attribute value definitions for user-defined actor attributes can be specified. These forms are similar to those available for built-in and user-defined attributes for graph components (see Sections 3.7 and 3.8).

In Figure 11, the square brackets ('[' and ']') are not part of the DIF syntax but are used instead to indicate that the `attributeType` specifier is *optional*. The `attributeType` specifier can be used to specify the type of the associated actor attribute. DIF supports three built-in actor attribute types: `INPUT`, `OUTPUT`, and `PARAMETER` to indicate the interface connections and parameters of an actor. These types are useful, for example, converting a dataflow graph to DIF from a particular dataflow-based design tool. In the syntax illustrated in Figure 11, DIF also allows users to specify an arbitrary (user-defined) identifier as the `attributeType`. In this way, the user can partition the attributes of an actor into a set of bins (corresponding to the attribute types) that include any combination of built-in actor attribute types and user-defined actor attribute types.

An attribute value for an actor can be assigned as 1) a value from one of the available DIF value types discussed in Section 3.10, 2) an identifier for specifying an associated graph component (edge, port, or parameter), or 3) a list of identifiers for indicating multiple associated graph components.

Section 6 and Section 7 contain more details and examples about how to use actor blocks.

```
actor nodeID {
   computation = "stringDescription";
   attributeID : [attributeType] = value;
   attributeID : [attributeType] = ID;
   attributeID : [attributeType] = ID1, ID2, ..., IDn;
}
```

Figure 11. Illustration of an `actor` block.

3.10 The Value Types

DIF supports most commonly used value types in DSP operations: integer, double, complex, integer matrix, double matrix, complex matrix, string, boolean, and array. Scientific notation is supported in DIF in the double format.

3.10.1 Integer

An integer value can be specified with an optional leading minus sign to indicate a negative value or zero (`-0`), or an optional leading plus sign to indicate a non-negative value. In the absence of a leading sign character, the integer value is interpreted as a non-negative value. Examples of integer value specifications are:

- `123`
- `-456`
- `+2`

3.10.2 Double

A double value can be formatted in various forms, as illustrated in the following examples:

- `123.456`
- `+0.1`
- `-3.6`
- `+1.2E-3`
- `-4.56e+7`

3.10.3 Complex

A complex value is enclosed by parentheses as $(\textit{real part}, \textit{imaginary part})$, and the real and imaginary parts are double values. For example, the complex value $(1.2\text{E}{-3} - j4.56\text{E}{+7})$ can be represented in DIF as $(1.2\text{E}{-3}, -4.56\text{E}{+7})$. If the definition of a complex value uses an integer value, as in $(-3, 4.5)$, then the integer is converted to a double before being stored in the corresponding part (real or imaginary) of the complex number.

3.10.4 Generality of Numeric Types

The three supported numeric types form a natural hierarchy in terms of generality: the complex type is the most general, the integer type is the least general, and the double type is in-between. This concept of generality is relevant when values of different types are mixed and implicit type conversions occur, as in the definition of matrices (see Section 3.10.5).

3.10.5 Matrix

Matrices are enclosed by square brackets; commas are used within a matrix to separate successive elements in the same row; and semicolons are used to separate successive rows. For example, the following illustrate definitions of 2×2 integer, double, and complex matrices, respectively:

```
[1, 2; 3, 4]
[+1.2, -3.4; -0.56e+7, 7.8E-3]
```

```
[(1.0, 2.0), (3.0, 4.0); (+1.2, -3.4), (-0.56e+7, 7.8E-3)]
```

If a matrix definition uses elements of different numeric values types, then the most general type that is used is taken to be the type of the matrix, and all elements in the matrix are stored in the form of this most general type. Thus, for example, the following two matrix definitions are equivalent (both define equivalent 2×3 double matrices):

```
[1, 2, 3; 4.2, 5, 6]
[1.0, 2.0, 3.0; 4.2, 5.0, 6.0]
```

3.10.6 String

String values are enclosed in double quotes. For example, `"FFT.c"` and `"FIR.asm"` could be string values that represent DSP library functions that correspond to an FFT and FIR actor, respectively. Such string values might be used, for example, in attribute value assignments for the built-in `computation` attribute of certain actors in a graph.

DIF supports escape sequences within strings. The support for escape characters in DIF is similar to that in C. The escape sequences supported in DIF are summarized in Figure 12.

DIF also supports the string concatenation operator ‘+’ to concatenate multiple strings — for example,

Escape sequence	Meaning
\’	Single quote
\”	Double quote
\\	Backslash
\n	Newline
\r	Carriage return
\t	Tab
\f	Form feed
\b	Backspace
$\backslash o_1 o_2 o_3$	Octal number
$\backslash \mathrm{x} h_1 h_2$	Hexadecimal number

Figure 12. The escape sequences supported in DIF strings.

```
“DIF is the:\n” + “dataflow interchange format”
```

is equivalent to

```
“DIF is the:\ndataflow interchange format”.
```

3.10.7 Boolean

A Boolean value is either `true` or `false`; these two keywords are used to represent Boolean values in DIF.

3.10.8 Array

An array based on any of the aforementioned value types is expressed inside braces. Each element is separated by “,” and can be any of the aforementioned value types.

The following are examples of Boolean, string, and matrix arrays, respectively:

```
{true, true, false, false, true, false}
{“FIR-embedded.c”, “FIR-host.java”, “FIR-FPGA.v”}
{[1, 2; 3, 4], [10, 11; 12, 13], [-2, -1; 0, 1]}
```

The following is an example of a *heterogeneous array*, which is an array that involves different value types.

```
{(3. 0, -1.5), “invalid”, -9, [2, 2; 1, 4], [3, 3, 1; 1, 1, 1]}
```

This array contains five elements — a complex number, a string, an integer, a 2×2 matrix, and a 2×3 matrix.

In some contexts, DIF arrays are perhaps most naturally viewed as *lists*; however, in DIF language terminology, the construction is consistently referred to as an *array*.

3.10.9 Unsupported types

The data types that are supported in DIF are chosen to cover the types that are used most commonly in signal processing applications. If a certain data type is needed for an application of DIF, but is not supported in DIF, it can often be handled to some extent by representation through the string type. Such use of the string type is useful, for example, in handling unsupported type issues in some interchange applications.

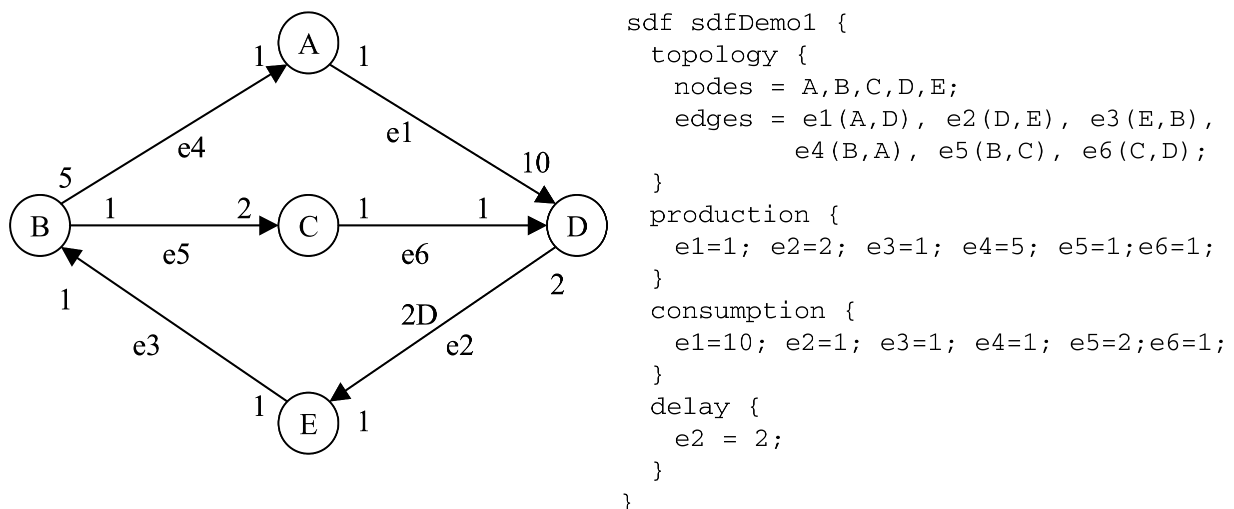
4 Dataflow Models

The DIF language is designed to specify arbitrary dataflow-oriented models of computation for DSP and other streaming-related application domains. Thus, its syntax and other features should be capable of describing a wide range of dataflow semantics. The present version of DIF provides direct support for a variety of dataflow models, including cyclo-static dataflow (CSDF), synchronous dataflow (SDF), single-rate dataflow, homogeneous synchronous dataflow (HSDF), and more complicated dataflow semantics such as boolean dataflow (BDF). Support for the dataflow-oriented meta-modeling technique of parameterized dataflow is also included. This section reviews the forms of dataflow that are supported directly in DIF, and provides examples that illustrate how to specify DIF graphs in terms of these models.

4.1 Synchronous Dataflow

Synchronous dataflow (SDF) [5, 16] is the most popular form of dataflow modeling for DSP system design. SDF permits the number of tokens produced and consumed by an actor to be a positive integer, which makes it suitable for modeling *multi-rate* DSP systems. However, SDF also imposes the restriction that all production and consumption rates must be fixed and known at compile-time. An edge e in an SDF graph has three integer-valued attributes, $prd(e)$, $cns(e)$, and $delay(e)$, which specify, respectively, the number of tokens produced by $src(e)$ onto e, the number of tokens consumed by $snk(e)$ from e, and the delay associated with e. According to pure SDF semantics, the first two of these attributes must be positive-valued, while the delay must be non-negative. The restriction that production and consumption rates are constant provides for various useful features of SDF, including static scheduling; decidable verification of bounded memory and deadlock-free operation; various forms of predictability; and a wide variety of useful optimization techniques (e.g., see [6]). These features come at the expense of limited expressive power — in particular, due to lack of support for data-dependent actor interface behavior.

The *dataflowModel* keyword for SDF is `sdf`. The edge attributes $prd(e)$, $cns(e)$, and $delay(e)$, are specified, respectively, for SDF graphs using the built-in attributes `production`, `consumption`, and `delay`. Figure 13 illustrates a simple SDF example in DIF.




```
sdf sdfDemo1 {
  topology {
    nodes = A,B,C,D,E;
    edges = e1(A,D), e2(D,E), e3(E,B),
            e4(B,A), e5(B,C), e6(C,D);
  }
  production {
    e1=1; e2=2; e3=1; e4=5; e5=1;e6=1;
  }
  consumption {
    e1=10; e2=1; e3=1; e4=1; e5=2;e6=1;
  }
  delay {
    e2 = 2;
  }
}
```

Figure 13. An SDF example and the corresponding DIF specification.

4.2 Single-rate Dataflow and Homogeneous Synchronous Dataflow

Single-rate dataflow graphs are SDF graphs in which all actors execute at the same average rate. As a result, the number of tokens produced on an edge when the source node fires is always equal to the number of tokens consumed on the same edge when the sink node fires. In other words, for every edge e in a single-rate dataflow graph, $prd(e) = cns(e)$. The *dataflowModel* keyword for single-rate dataflow is `singlerate`. Because all nodes execute at the same average rate, DIF uses the built-in attribute `transfer` to specify token transfer rates for single-rate graphs instead of `production` and `consumption` attributes. The value of the transfer attribute for an edge e is equal to the common value of $prd(e)$ and $cns(e)$.

In *homogeneous synchronous dataflow* (HSDF), the production rate and consumption rate are restricted to be unity on all edges. HSDF can be viewed as a restricted case of single-rate dataflow and SDF. The *dataflowModel* keyword for HSDF is `hsdf`. Because of the homogeneous unit transfer rate that is implicit for all edges in an HSDF graph, specifying `production`, `consumption`, or `transfer` values is not necessary in DIF specifications of HSDF graphs.

Single-rate and HSDF graphs are useful models in scenarios such as uniform execution rate processing, precedence expansion for multi-rate SDF graphs, and multiprocessor scheduling. Utilities for converting among SDF, single-rate, and HSDF graphs, based on methods introduced in [16], are provided in DIF. Such conversion is illustrated in Figure 14. Detailed coverage of these methods and their application is provided in [20].

4.3 Cyclo-static Dataflow

In *cyclo-static dataflow* (CSDF) [7], a production rate or consumption rate is allowed to vary as long as the variation forms a fixed, periodic pattern. More precisely, each actor A in a CSDF graph is associated with a fundamental period $\tau(A) \in Z^{+}$, which specifies the number of phases in one minimal period of the cyclic production / consumption pattern of A. Each time an actor is fired in a period, a different phase is executed. For each incoming edge e of A, $cns(e)$ is specified as a $\tau(A)$-tuple ($C_{e,1}$, $C_{e,2}$, ..., $C_{e,\tau(A)}$), where each $C_{e,i}$ is a non-negative integer that gives the number of tokens consumed from e by A in the i-th phase of each period of A. Similarly, for each outgoing edge e of A, $prd(e)$ is specified as a $\tau(A)$-tuple ($P_{e,1}$, $P_{e,2}$, ..., $P_{e,\tau(A)}$), where each $P_{e,i}$ is a non-negative integer that gives the number of tokens produced onto e by A in the i-th phase. CSDF offers more flexibility in representing interactions between actors and scheduling, but its expressive power at the level of overall individual actor functionality is the same as SDF. Further details on the features of CSDF are discussed in [4, 7, 17].

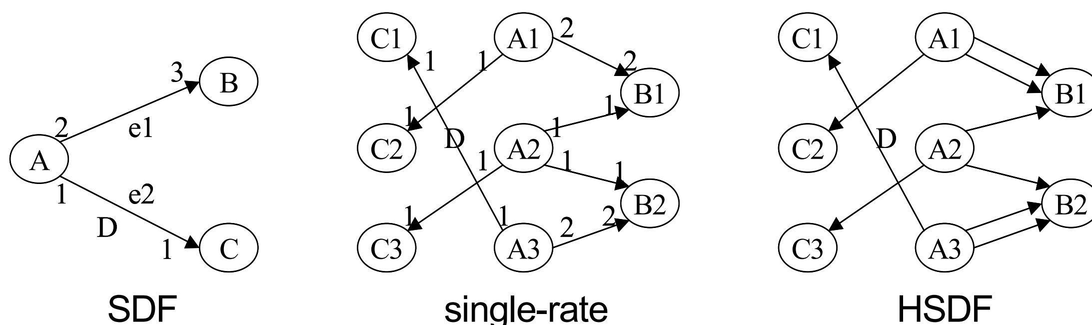


Figure 14. Conversion among SDF, single-rate, and HSDF graphs.

The *dataflowModel* keyword for CSDF is `csdf`. For a CSDF graphs in DIF, the built-in attributes `production` and `consumption` are specified as $\tau(A) \times 1$ integer matrices (column vectors) that represent the associated $\tau(A)$-tuple patterns for each fundamental period of the relevant actor A. Figure 15 illustrates an example of a CSDF specification in DIF. This example involves actors for input/output interfacing, upsampling, downsampling, and FIR filtering.

4.4 Boolean-controlled dataflow

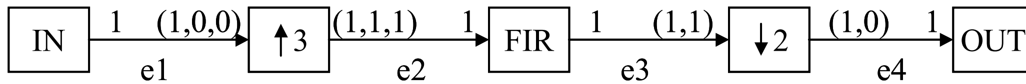
Boolean-controlled dataflow (BDF) [8] is a form of dynamic dataflow for supporting data-dependent DSP computations while still permitting quasi-static scheduling to a certain degree. BDF is Turing-complete [8]. Quasi-static scheduling refers to a form of scheduling in which a significant proportion of scheduling decisions is made at compile-time through analysis of static properties in the application model. By including BDF, DIF improves its ability to work with Turing-complete semantics and incorporates detailed support for an important, fully expressive model.

In dynamic dataflow modeling, a *dynamic actor* produces or consumes certain numbers 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 designated Boolean “control token.” In other words, the number of tokens that a dynamic BDF actor A produces on an edge e_o or consumes from an edge e_i during each firing is determined by the Boolean value of the control token consumed by A during that firing. Here $e_o \in out(A)$ or $e_i \in in(A)$.

BDF imposes the restriction that a dynamic actor can only consume one control token during each firing. The following two equations describe intuitively the structure of dynamic production and consumption rates in BDF.

$$prd(e_o) \text{ at i-th iteration} = \begin{cases} \text{prod rate1, if control token consumed by A at i-th iteration is TRUE} \\ \text{prod rate2, if control token consumed by A at i-th iteration is FALSE} \end{cases}$$

$$cns(e_i) \text{ at i-th iteration} = \begin{cases} \text{cons rate1, if control token consumed by A at i-th iteration is TRUE} \\ \text{cons rate2, if control token consumed by A at i-th iteration is FALSE} \end{cases}$$




```
csdf csdfDemo1 {
  topology {
    nodes = IN, UP3, FIR, DOWN2, OUT;
    edges = e1(IN, UP3), e2(UP3, FIR), e3(FIR, DOWN2), e4(DOWN2, OUT);
  }
  production {
    e1=1; e2=[1,1,1]; e3=1; e4=[1,0];
  }
  consumption {
    e1=[1,0,0]; e2=1; e3=[1,1]; e4=1;
  }
}
```

Figure 15. A CSDF example and the corresponding DIF specification.

Actors in BDF graphs that are not dynamic (Boolean-controlled) are called *regular* actors. The term *regular* in this BDF context is synonymous with *SDF* — that is, regular actors in BDF produce and consume constant amounts of data.

The *dataflowModel* keyword for BDF is `bdf`. The built-in attributes `production` and `consumption` can be used to specify both fixed (regular) and dynamic (Boolean-controlled) production and consumption rates. For a fixed rate, the syntax is the same as SDF; for a Boolean-controlled rate, a 2×1 integer matrix (column vector) is used to specify the two relevant Boolean-controlled values. The first element in this matrix gives the associated rate when the control token is *TRUE*, and the second element gives the rate when the control token is *FALSE*.

Figure 16 illustrates a BDF example that implements an *if-else* statement. This example includes two commonly-used BDF dynamic actors, *SWITCH* and *SELECT.* The SWITCH actor consumes one token from its incoming edge and copies that token to either a "true" outgoing edge or a "false" outgoing edge according to the value of its control token. The SELECT actor consumes one token T from either a "true" incoming edge or a "false" incoming edge according to the value of its control token, and then copies T to the outgoing edge.

4.5 Parameterized Synchronous Dataflow

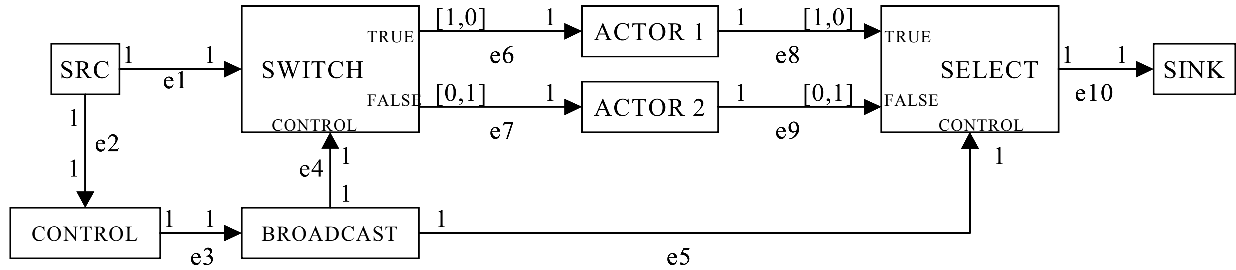
Parameterized dataflow modeling differs from various other fundamental dataflow modeling techniques such as SDF, CSDF, in that it is a *meta-modeling* technique. Parameterized dataflow can be applied to any underlying "base" dataflow model that has a well-defined notion of a *graph iteration*. Applying parameterized dataflow in this way augments the base model with powerful capabilities for dynamic reconfiguration and quasi-static scheduling through parameterized looped schedules [2]. Combining parameterized dataflow with synchronous dataflow forms *parameterized synchronous dataflow* (PSDF), a dynamic dataflow model that has been investigated in depth and shown to have useful properties [2].

A PSDF actor A is characterized by a set $params(A)$ of parameters that can control the actor's functionality, including the actor's dataflow behavior, such as its production rates and consumption rates. A configuration $config_A$ of a PSDF actor A is determined by assigning values to the parameters of A. In a PSDF specification, each actor parameter is either assigned a value or left unspecified. The statically-unspecified parameters in a PSDF specification are assigned values at run time, thus dynamically configuring the actor's functionality. Furthermore, a given parameter can have its value changed an arbitrary number of times as a PSDF system executes, thus providing for flexible dynamic reconfiguration of actor functionality.

Given a PSDF graph G, all statically-unspecified actor parameters in G propagate "upwards" in the enclosing specification hierarchy as parameters of the PSDF graph G, which are denoted as $params(G)$. A DSP application is usually modeled in PSDF through a *PSDF specification*, which is also called a *PSDF subsystem*. A PSDF subsystem Φ consists of three PSDF graphs, the *init graph* Φ_i, the *subinit graph* Φ_s, and the *body graph* Φ_b. Generally, the body graph models the main functional behavior of the subsystem, whereas the init and subinit graphs control the behavior of the body graph by appropriately configuring parameters of the body graph. Moreover, PSDF employs a hierarchical modeling structure by allowing a PSDF subsystem Φ to be embedded in a "parent" PSDF graph G and abstracted as a hierarchical PSDF actor H, where $\Phi = subsystem(H)$.

The init graph Φ_i does not take part in the interface dataflow of Φ, and also, all parameters of Φ_i are left unspecified (for configuration by higher-level subsystems). The subinit graph Φ_s may only accept dataflow inputs at its interface input ports, and each parameter of Φ_s is configured

either by an interface output port of Φ_i, is set by an interface input port of Φ, or is left unspecified. The interface output ports of Φ_i and Φ_s are reserved exclusively for configuring parameter values. The body graph Φ_b usually takes on the major role in dataflow processing and all of its




```
if CONTROL outputs TRUE token
   fire ACTOR1;
else
   fire ACTOR2;
end
```

```
bdf bdfDemo1 {
  topology {
    nodes = SRC, SWITCH, SELECT, SINK, CONTROL, BROADCAST, ACTOR1, ACTOR2;
    edges = e1(SRC,SWITCH), e2(SRC,CONTROL), e3(CONTROL,BROADCAST),
            e4(BROADCAST,SWITCH), e5(BROADCAST,SELECT), e6(SWITCH,ACTOR1),
            e7(SWITCH,ACTOR2), e8(ACTOR1,SELECT), e9(ACTOR2,SELECT),
            e10(SELECT,SINK);
 }
  production {
    e1=1; e2=1; e3=1; e4=1; e5=1; e6=[1,0]; e7=[0,1]; e8=1; e9=1; e10=1;
  }
  consumption {
    e1=1; e2=1; e3=1; e4=1; e5=1; e6=1; e7=1; e8=[1,0]; e9=[0,1]; e10=1;
  }
  actor SWITCH {
    computation; = “dif.bdf.SWITCH“;
    control : CONTROL = e4;
    input : INPUT = e1;
    true : TRUEOUTPUT = e6;
    false : FALSEOUTPUT = e7;
  }
  actor SELECT {
    computation; = “dif.bdf.SELECT“;
    control : CONTROL = e5;
    output : OUTPUT = e10;
    true : TRUEINPUT = e8;
    false : FALSEINPUT = e9;
  }
```

Figure 16. A BDF example, the corresponding pseudocode, and the DIF specification.

dynamic parameters are configured by the interface output ports of Φ_i and Φ_s. All unspecified parameters of Φ_i and Φ_s propagate "upwards" as the subsystem parameters of Φ, which are denoted as $params(\Phi)$, and are configured by the init and subinit graphs of hierarchically higher level subsystems. This mechanism of parameter configuration is referred to as *initflow*.

In order to maintain a valuable level of predictability and efficient quasi-static scheduling, PSDF requires that the interface dataflow of a subsystem must remain unchanged throughout any given iteration of its hierarchical parent subsystem. Therefore, parameters that determine the interface dataflow can only be configured by output ports of the init graph Φ_i, and Φ_i is only invoked once at the beginning of each invocation of the supergraph (parent graph). As a result, the parent has a consistent view of module interfaces throughout any iteration. On the other hand, parameter reconfiguration that does not change interface behavior of subsystem is permitted to occur across iterations of the subsystem rather than the parent subsystem. The subinit graph Φ_s performs this reconfiguration activity and is invoked each time before an invocation of the body graph Φ_b. This gives a subsystem a consistent view of its components' configurations throughout any given iteration and provides configurability across iterations.

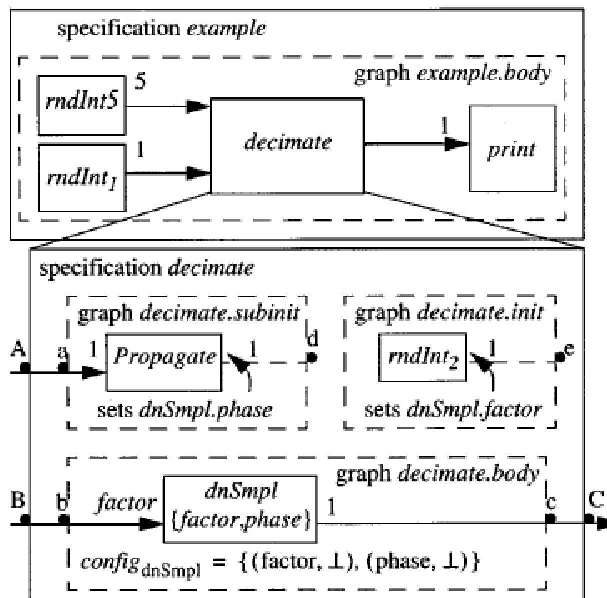
DIF separates PSDF graphs and PSDF subsystems into two different modeling blocks, and the corresponding *dataflowModel* keywords for them are `psdf` and `psdfSubsystem`, respectively. The `parameter` block of DIF is well-suited for helping to specify PSDF-based designs. Configurable actor attributes and non-static dataflow modeling attributes, such as production rates and consumption rates are parameterized by pre-defined DIF parameters. Unspecified parameters are defined without providing their values in the parameter block. *Upward parameters* of a PSDF subsystem (i.e., parameters that propagate hierarchically upward, as discussed previously) can be specified in the `refinement` block of its supergraph. For hierarchical modeling structures in PSDF, such as those used when PSDF specifications are nested, the DIF hierarchy concepts described in Section 2.2 can fully represent the associated functionality, and the DIF `refinement` block is used to specify such structures.

DIF interprets a PSDF subsystem as a graph that consists implicitly of three subgraphs, Φ_i, Φ_s, and Φ_b. In a DIF specification, a PSDF subsystem does not need a `topology` block because the three subgraphs — init, subinit, and body (Φ_i, Φ_s, and Φ_b) — are built-in and there is no edge connection in any PSDF subsystem. More precisely, a DIF specification for a PSDF subsystem should not contain a `topology` block. On the other hand, a DIF specification for a PSDF graph may (and usually does) contain a `topology` block. Parameter configuration across init, subinit, and body graphs is specified using the built-in attribute `paramConfig` with the syntax illustrated in Figure 18.

Figure 17 illustrates a PSDF example from [2] and the corresponding DIF specification.

```
paramConfig {
   subinitGraphID.paramID = initGraphID.outputPortID;
   bodyGraphID.paramID = initGraphID.outputPortID;
   bodyGraphID.paramID = subinitGraphID.outputPortID;
}
```

Figure 18. Illustration of the DIF syntax for parameter configuration in PSDF.




```
psdf decimateSubinit {
  topology { nodes = Propagate; }
  interface {
    inputs = a:Propagate;
    outputs = d:Propagate;
  }
  consumption { a = 1; }
  production { d = 1; }
}

psdf decimateInit {
  topology { nodes = rndInt2; }
  interface { outputs = e:rndInt2; }
  production { e = 1; }
}

psdf decimateBody {
  topology { nodes = dnSmpl; }
  interface {
    inputs = b:dnSmpl;
    outputs = c:dnSmpl;
  }
  parameter {
    factor;
    phase;
  }
  consumption { b = factor; }
  production { c = 1; }
}
```

```
psdfSubsystem decimate {
  interface {
    inputs = A:subinit, B:body;
    outputs = C:body;
  }
  refinement { decimateInit = init; }
  refinement {
    decimateSubinit = subinit;
    a : A;
  }
  refinement {
    decimateBody = body;
    b : B;
    c : C;
  }
  paramConfig {
    decimateBody.factor =
        decimateInit.e;
    decimateBody.phase =
        decimateSubinit.d;
  }
}

psdf exampleBody {
  topology {
    nodes = rndInt5, rndInt1,
            decimate, print;
    edges = e1(rndInt5, decimate),
            e2(rndInt1, decimate),
            e3(decimate, print);
  }
  refinement {
    decimate = decimate;
    A : e1; B : e2; C : e3;
  }
  production {
    e1 = 5; e2 = 1;
  }
  consumption { e3 = 1; }
}

psdfSubsystem example {
  refinement { decimate = body; }
}
```

Figure 17. A PSDF example and the corresponding DIF specification.

4.6 Binary Cyclo-static Dataflow

Binary CSDF (BCSDF) is a restricted form of CSDF in which production and consumption rates are constrained to be binary vectors. In other words, elements of BCSDF production and consumption vectors are either 0 or 1. BCSDF graphs arise naturally, for example, when converting certain kinds of process networks to dataflow (e.g., see [9]) and when modeling dataflow-based hardware implementations.

The *dataflowModel* keyword for BCSDF is `bcsdf`. Other than the use of this keyword and the restriction to binary production and consumption rate vectors, the DIF specification format for BCSDF is as the same as that for CSDF. In some BCSDF representations, the numbers of actor phases can be very large. Therefore, the BCSDF intermediate representation format of the DIF package utilizes bit vectors to store BCSDF production and consumption rates.

4.7 Interval-Rate Locally-static Dataflow

Interval-rate, locally-static dataflow (ILDF) [23] has been developed to represent and analyze a class of dataflow graphs in which production and consumption rates are not known precisely at compile time. In ILDF graphs, the production and consumption rates remain constant throughout execution (locally-static), however, the values of these constants are not necessarily known at compile time. Instead, for each edge e, there are intervals $[p_{min}, p_{max}]$ and $[c_{min}, c_{max}]$ that give minimum and maximum values for the production and consumption rates, respectively, associated with e.

DIF is capable of representing ILDF graphs by parameterizing the production and consumption rates in a graph, and specifying the intervals of those parameters using the facilities in DIF for specifying parameter ranges (Section 3.5). Figure 19 illustrates an ILDF example and a corresponding DIF specification.

5 The DIF Package

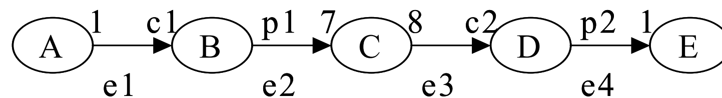
The DIF package is a Java-based software package that is developed along with the DIF language. In general, the DIF package consists of three major parts: the DIF front-end, the DIF representation, and the implementations of dataflow-based analysis, scheduling, and optimization algorithms. This section introduces the major parts of the DIF package and describes the relationship of the DIF package to theoretical dataflow models, dataflow-based DSP design tools, and their underlying embedded processing platforms.

5.1 The DIF Representation

For each supported dataflow model of computation, the DIF package provides an extensible set of data structures (object-oriented Java classes) for representing and manipulating dataflow graphs in the model. This graph-theoretic intermediate representation for a dataflow model is usually referred to as the *DIF representation* for the model.

The DIFGraph is the most general graph class in the DIF package. It represents the basic dataflow graph structure among all dataflow models and provides methods for manipulating graphs that are common to all models supported in DIF. For a more specialized dataflow model, development can proceed naturally by extending the general DIFGraph class (or some suitable subclass), and overriding and adding new methods to perform more specialized functions.

Figure 20 shows the class hierarchy of graph classes in the DIF package. The DIFGraph is extended from the DirectedGraph class, and in turn, from the Graph class. The DirectedGraph and




```
ildf ildfDemo1 {
  topology {
    nodes = A, B, C, D, E;
    edges = e1(A,B), e2(B,C), e3(C,D), e4(D,E);
  }
  parameter {
    c1 : [3,7];
    c2 : [3,7];
    p1 : [2,10];
    p2 : [2,10];
  }
  production {
    e1 = 1; e2 = p1; e3 = 8; e4 = p2;
  }
  consumption {
    e1 = c1; e2 = 7; e3 = c2; e4 = 1;
  }
}
```

Figure 19. An ILDF example and the corresponding DIF specification.

Graph classes are used from the publicly-available `mocgraph` package, which provides data structures and methods for manipulating generic graphs, especially generic graphs that arise in the context of various models of computation ("mocs"). The dataflow models CSDF, SDF, single-rate dataflow, and HSDF are related in such a way such that each succeeding model among these four is a special case of the preceding model. Accordingly, CSDFGraph, SDFGraph, SingleRateGraph, and HSDFGraph form a class hierarchy in the DIF package such that each succeeding graph class inherits from the more general one that precedes it (see Figure 20).

In addition to the aforementioned fundamental dataflow graph classes, the DIF package also provides the BDFGraph representation for the Turing complete BDF model; the PSDFGraph representation for modeling of dataflow graph reconfiguration; and the BCSDFGraph representation for the specialized BCSDF model. Furthermore, a variety of other dataflow models are being explored for inclusion in later versions of DIF.

5.2 The DIF Front-end

Although the DIF language is able to specify a wide variety of dataflow models, in practice, the DIF representation is the actual format for realizing dataflow graphs and for performing analysis, scheduling, and optimization when working with the DIF package. Thus, automatic conversion between DIF specifications (.dif files) and DIF representations (graph instances) is a fundamental feature of the DIF package. The DIF front-end tool automates this conversion and provides users an integrated set of programming interfaces to construct DIF representations from specifications and to generate DIF specifications from intermediate representations.

The DIF front-end consists of a `Reader` class, a set of language parsers (`LanguageAnalysis` classes), a `Writer` class, and a set of graph writer classes. The language parsers are implemented using a Java-based compiler compiler called `SableCC` [12]. The flexible structure and Java integration of the `SableCC` compiler enables easy extensibility for parsing different dataflow graph types.

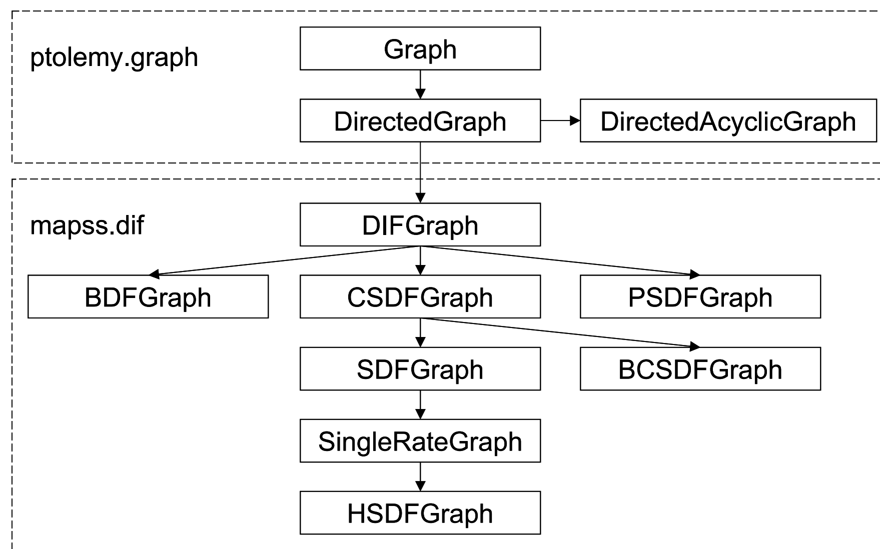


Figure 20. The class hierarchy of graph types (*DIF representations*) in the DIF package.

Figure 21 illustrates how the DIF front-end constructs a DIF representation (graph class) according to a given DIF specification. The `Reader` class invokes the corresponding language analysis class (DIF language parser) based on the model keyword specified in the DIF specification. Then, the language analysis class constructs a graph instance according to the dataflow semantics specified in the DIF specification.

On the other hand, Figure 22 illustrates how the DIF front-end generates a DIF specification according to a DIF representation. The `Writer` class invokes the corresponding graph writer class based on the type of the given graph instance. After that, the graph writer class generates the DIF specification by tracing elements and attributes of the graph instance.

In the DIF package, the language analysis classes (language parsers) are used for parsing the DIF language. The main differences between these classes are in their processing of built-in attributes and their instantiation of DIF representations. Similarly, the graph writer classes are used for writing out the dataflow semantics in DIF form, and these classes differ from one another mainly in how they handle built-in attributes and the *dataflowModel* keyword.

All specialized dataflow language analysis classes are extended from the `LanguageAnalysis` class that constructs the `DIFGraph` representation, which is the most general model supported in DIF. Likewise, all specialized graph writer classes are extended from the `DIFWriter` class, which writes out the dataflow semantics of `DIFGraph` instances. Typically, an extended class overrides only a small set of model-specific methods.

5.3 Dataflow-based Algorithms for Analysis, Scheduling, and Optimization

For supported dataflow models, the DIF package provides not only graph-theoretic intermediate representations but also efficient implementations of various useful analysis, scheduling, and optimization algorithms that operate on the representations. By building on the DIF representations and existing algorithm implementations, and invoking the built-in algorithms as needed, emerging techniques and other new algorithm implementations can conveniently be developed and experimented with through the DIF package.

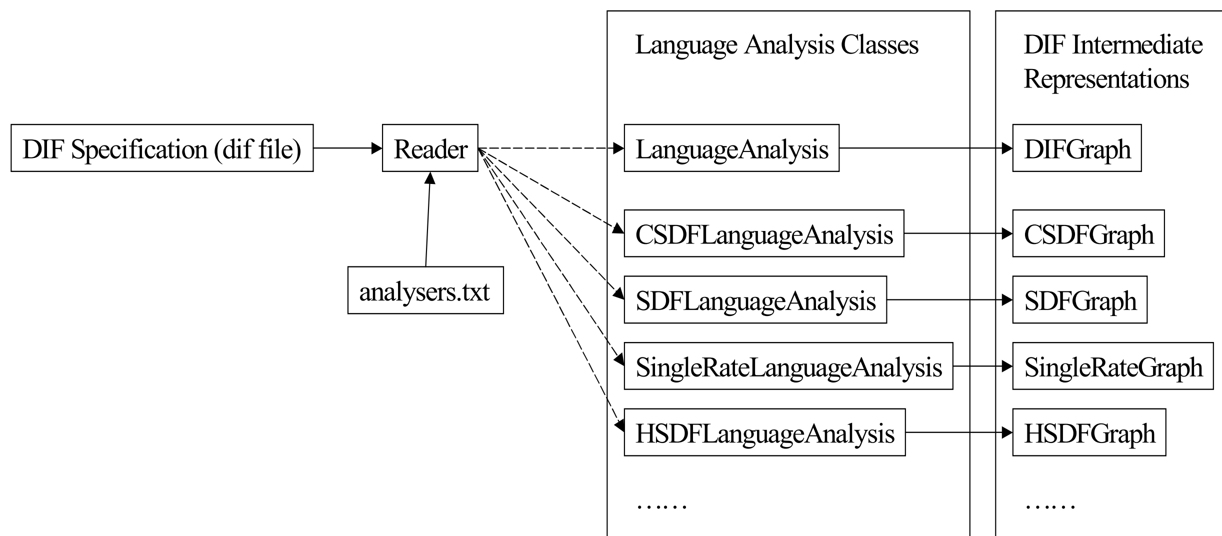


Figure 21. The DIF Front-end: from DIF specification to DIF representation.

5.4 Methodology of using DIF

Figure 23 illustrates the conceptual architecture of DIF and key relationships among abstract dataflow models, dataflow-based DSP design tools, DIF specifications, and the DIF package. The block in this diagram labeled *Dataflow Models* represents the dataflow models currently supported in DIF. Based on the DIF language introduced in Section 3, application models using these dataflow models can be specified as DIF specifications, which are described in Section 4.

The block labeled *Dataflow-based DSP Design Tools* represents the set of tools for DSP system design that are currently available, and other previously-developed DSP design tools. These tools usually provide a block-diagram-based graphical design environment, a set of libraries consisting of useful modules, and a programming interface for designing modules. As long as the DSP system modeling capability in a design tool is based on dataflow principles, the DIF language is able to capture the associated dataflow semantics and related modeling information of DSP applications in the tool and represent them in the form of DIF specifications.

The DIF package realizes the abstract dataflow structure of DSP application models through the DIF representation. With the DIF front-end tool, the DIF representation can be constructed automatically based on the given DIF specification. After that, dataflow-based analysis, scheduling, and optimization techniques can be applied on the DIF representation.

Figure 24 illustrates the implementation and end-user viewpoints of the DIF architecture. DIF supports a layered design methodology covering dataflow models, the DIF language, DIF specifications, the DIF package, dataflow-based DSP design tools, and the underlying hardware and software platforms targeted by these tools.

The *Dataflow Models* layer in Figure 24 represents the dataflow models currently integrated in the DIF package. These models can be further categorized into *static* dataflow models such as SDF and CSDF; *dynamic* dataflow models such as the Turing-complete BDF model; and *meta-modeling* techniques such as parameterized dataflow, which provides the dynamic reconfiguration capability of PSDF. Using the DIF language, application behaviors compatible with these data-

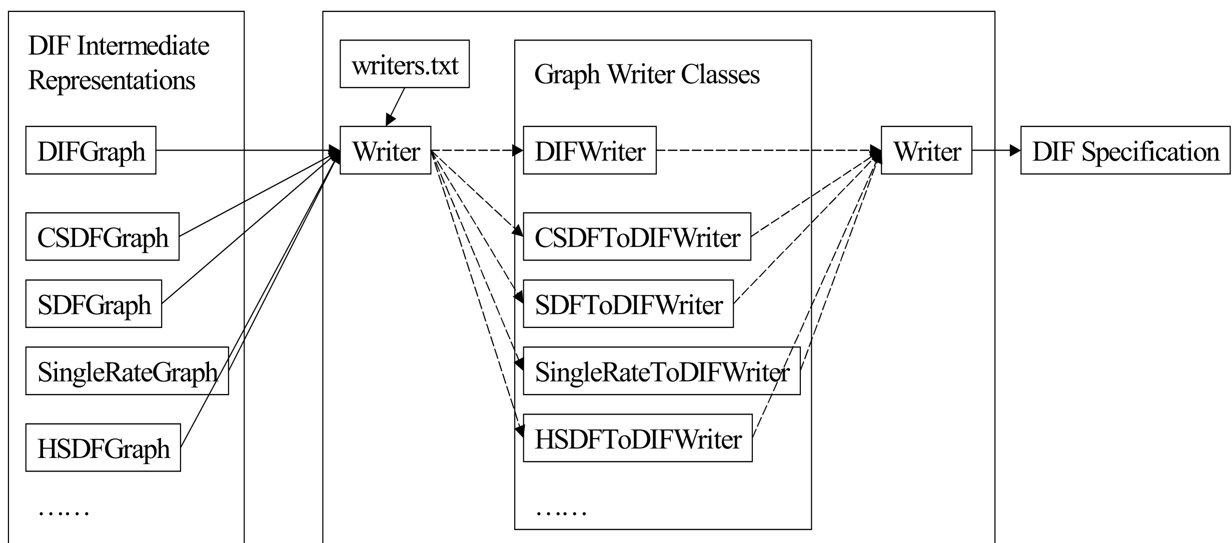


Figure 22. The DIF front-end: from DIF representation to DIF specification.

flow modeling techniques can be specified in a streamlined manner as specialized DIF specifications.

The primary dataflow-based DSP design tools that we experimented with in our initial development of DIF are the SDF domain of Ptolemy II [11], developed at UC Berkeley, and the Autocoding Toolset developed by MCCI. However, DIF is in no way designed to be specific to these tools; they have been used only as a starting point for experimenting with DIF in conjunction with sophisticated academic and industrial DSP design tools, respectively. Tools such as these form an important layer in our proposed DIF-based design methodology. Ptolemy II is a Java-based design environment and utilizes the Modeling Markup Language (MoML) [15] as its textual format for specification and interchange. Ptolemy II provides multiple models of computation, including some dataflow-based models and a variety of non-dataflow models, and a large set of libraries consisting of actors for various application domains. On the other hand, the MCCI Autocoding Toolset is based on Processing Graph Method (PGM) semantics [21], and uses Signal Processing Graph Notation (SPGN) as its specification format. It also provides an efficient library consisting of domain primitives for DSP computations, and is able to synthesize software implementations for certain high-performance platforms.

The layer in Figure 24 labeled *Hardware / Software Embedded Systems* generally represents all embedded platforms that are supported by dataflow-based DSP design tools. For example, Ptolemy II can generate executable Java code for the Java virtual machine (VM); and the Autoc-

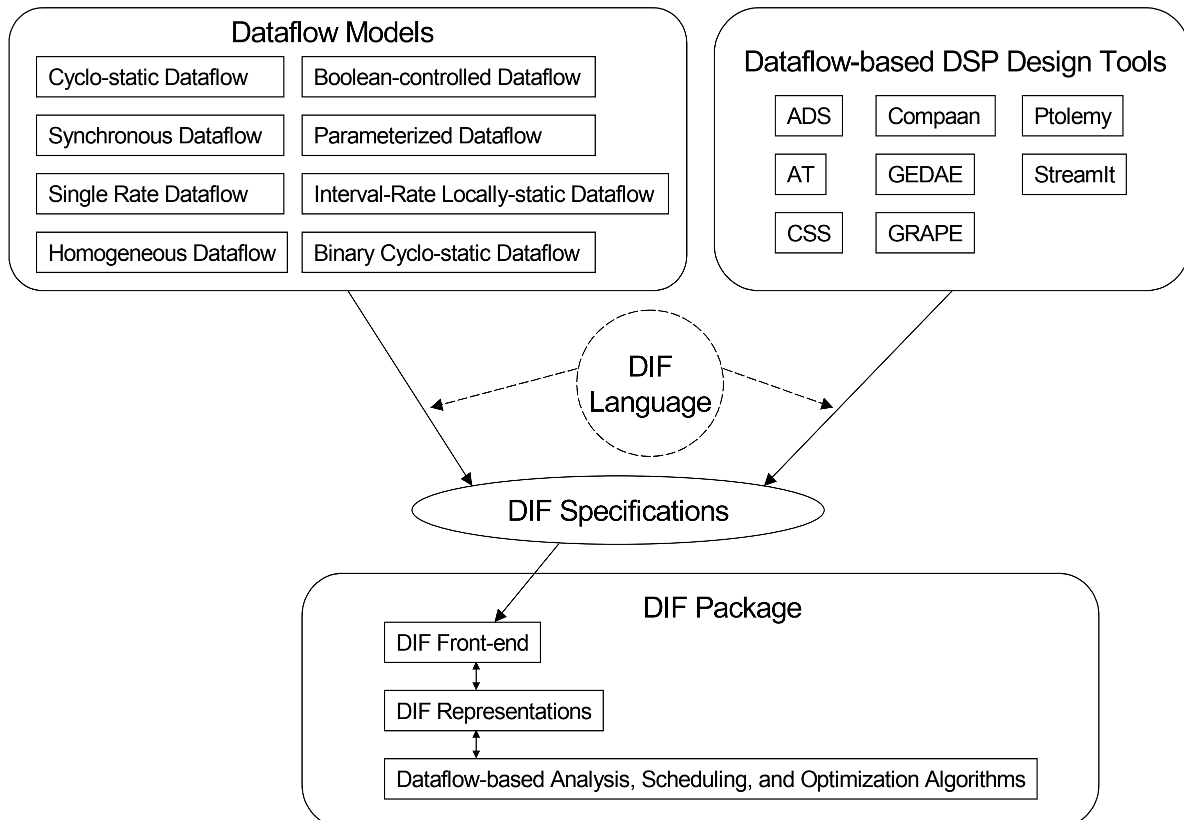


Figure 23. Relationships among dataflow models, design tools, the DIF language, DIF specifications, and the DIF package.

oding Toolset can generate executable C code for UNIX/LINUX based symmetric multiprocessors (SMP) and the Mercury family of embedded processors and Ada code for the Virtual Design Machine, UNIX/LINUX based networks utilizing TCP/IP based communications.

The DIF package provides an intermediate layer between abstract dataflow models and different kinds of practical implementations. In this role, DIF handles the concrete realization of dataflow graphs and application of dataflow-based algorithms to analyze these graphs. DIF exporting and importing tools automate the process of exporting DSP applications from design tools into equivalent DIF specifications and conversely, importing DIF representations into design tools. Automating the exporting and importing processes between DIF and design tools provides the DSP design industry a useful front-end to using DIF and the DIF package. In the next section, we will describe issues involved in such automation, and approaches taken in the DIF framework to address these issues.

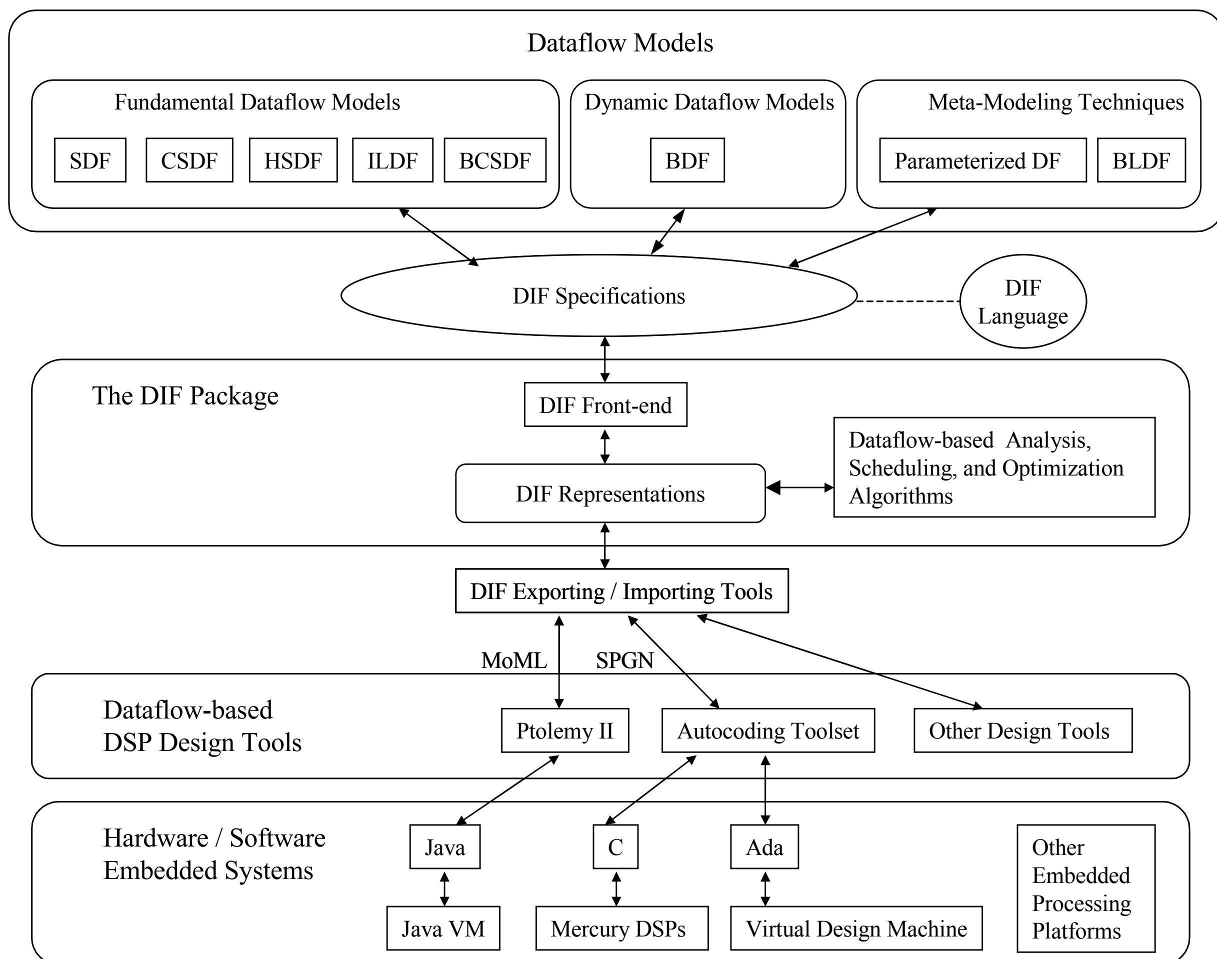


Figure 24. The role of DIF in DSP system design.

6 Exporting and Importing DIF

The DIF language is capable of specifying dataflow semantics of DSP applications from any dataflow-based design tool. When integrating features of DIF with a DSP design tool, incorporating capabilities to translate between the design tool's specification format and DIF specifications or DIF representations is usually an essential first step. In DIF terminology, *exporting* means translating a DSP application from a design tool's specification format to DIF (either to the DIF language or directly to the appropriate form of DIF representation). On the other hand, *importing* means translating a DIF specification to a design tool's specification format or converting a DIF representation to a design tool's internal representation format. Figure 25 illustrates the exporting and importing mechanisms between DIF and design tools.

Parsing design tool specification formats and then directly formulating the corresponding DIF specifications is usually not the most efficient way to go about exporting. Instead, it is useful to build on the complete set of classes that DIF provides for representing dataflow graphs in well-designed, object-oriented realizations. Hence, instead of parsing and directly formulating equivalent DIF language code, mapping graphical representations from design tools to DIF representations and then converting to DIF specifications, using representation-to-specification translation capabilities already built in to DIF is typically much easier and more efficient.

However, depending on the particular design tool involved, it still may be a somewhat involved task to automate the exporting and importing processes. First, graph topologies and hierarchical structures of DSP applications must be captured in order to completely represent their dataflow semantics. Furthermore, actor computations, actor parameters, and inter-actor connections must also be specified for preserving application functionality completely. In the following subsections, we explain more about these issues and describe our approaches to addressing them. For illustration, we also demonstrate exporting and importing capabilities that we have developed between DIF and Ptolemy II.

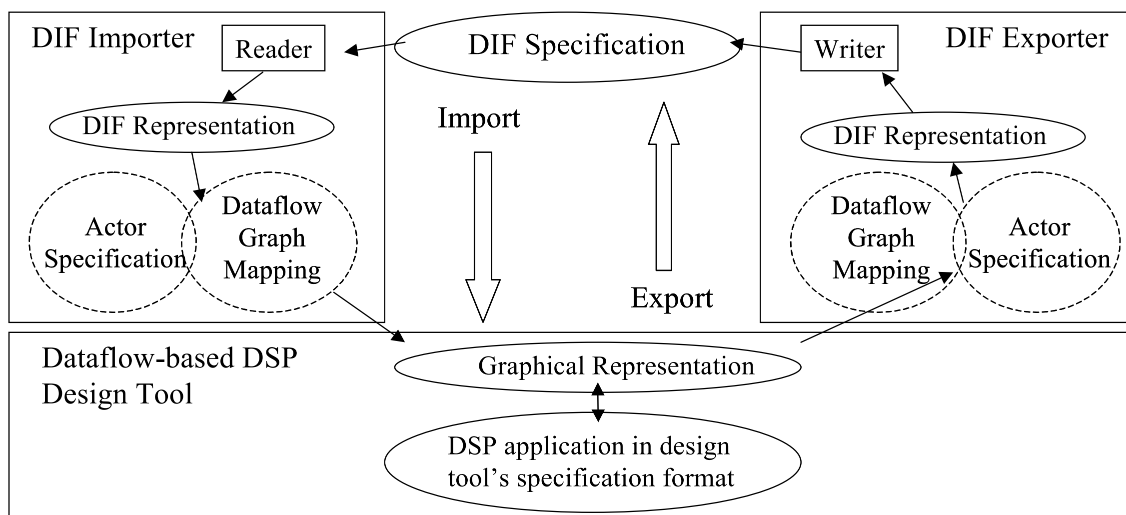


Figure 25. Exporting and Importing Mechanism.

6.1 Mapping Dataflow Graphs

Dataflow-based DSP design tools usually have their own representations for nodes, edges, hierarchies, etc. Moreover, they often use more specific components instead of just the abstract components found in formal dataflow representations. Implementation issues involved in converting the graphical representations of design tools to the formal dataflow representations used in DIF are categorized as *dataflow graph mapping* issues.

We examine exporting Ptolemy II to DIF as an example here to explain problems in dataflow graph mapping. Ptolemy II includes an *AtomicActor* class for representing DSP computations (associated with primitive dataflow nodes) and a *CompositeActor* class for representing subgraphs. It uses a *Relation* class instead of edges to connect actors. Each actor has multiple *IOPort*s and those *IOPort*s are connection points for *Relation*s. A *Relation* can have a single source but fork to multiple destinations. Regular *IOPort*s can accept only one *Relation* but Ptolemy II also allows *multiport IOPort*s that can accept multiple *Relation*s. Clearly, challenges arise when mapping Ptolemy II graphical representations to DIF representations. First, based on the formal definition of nodes in dataflow models, vertices in DIF graphs do not have ports to distinguish actor-level interfaces. Second, edges in formal dataflow graphs cannot support multiple destinations (i.e., multiple sink actors for the same edge) in contrast to the more flexible kind of connectivity provided by Ptolemy II *Relation*s. Third, the *multiport* property in Ptolemy II does not match directly with formal dataflow semantics, and even an interface port of a hierarchy (defined in Section 2.2) can only connect to one outer edge or port.

Although implementation problems in dataflow graph mapping are tool-specific, exporting without losing any essential modeling information is generally feasible due to the broad range of modeling capabilities offered through the features in DIF. First, the DIF language is capable of describing dataflow semantics regardless of the particular design tool used to enter an application model as long as the tool is dataflow-based. Second, DIF representations can fully realize the dataflow graphs specified by the DIF language. Based on these two properties, our general approach to exporting involves comprehensively traversing a given graphical representation in its native design tool, and mapping the modeling components encountered during this traversal into equivalent components or groups of components available for DIF representations. After that, our DIF front-end tool can write the DIF representations into textual DIF specifications.

A brief description follows to illustrate how this approach can be applied to mapping graphical representations from Ptolemy II into DIF representations. First, *AtomicActor*s are represented by DIF nodes and *CompositeActor*s are represented by DIF hierarchies. Single-source-single-destination *Relation*s are represented by DIF edges. For a multiple-destination *Relation*, a *fork* actor (which is described in Section 6.3) and several accompanying DIF edges are used to represent the Relation without losing any of its dataflow properties. The *IOPort*s and the corresponding connections associated with a Ptolemy II actor are specified in DIF as actor attributes. For a *multiport IOPort* in Ptolemy II, multiple connections can be listed as an actor attribute in DIF.

6.2 Specifying Actors

In dataflow analysis, a graph node may be viewed as a functional unit that has some sort of *weight* associated with it, and consumes and produces certain numbers of tokens when executing. Here, by a node weight we mean an arbitrary set of node-specific information (e.g., estimates for execution time or code size) that accompanies a node. Often, dataflow-based analysis and scheduling techniques are based on production rates, consumption rates, edge delays, and other more special-

ized edge information (e.g., the inter-processor communication cost associated with an edge if its source and sink are mapped to different processors in a multiprocessor target), as well as various kinds of node weight information. The detailed computation performed by a node is irrelevant to many dataflow-based analyses — in particular it is irrelevant to analyses in which all relevant aspects of the computation are abstracted into node weights and related kinds of edge information. However, the computation (such as an FFT operation) and computation-related attributes (such as the order of the FFT) associated with a node are essential during implementation. To avoid confusion between the viewpoints of nodes in dataflow analyses versus in hardware/software implementation, we henceforth use the term *node* for the former context, and we use the term *actor* to refer to a node that has a specified computation and possibly other implementation-related attributes associated with it.

Specifying an actor's computation as well as all necessary operational information is referred to as *actor specification*. It is an important issue in exporting and importing between DIF and design tools as well as in porting DSP applications across tools because every actor's functionality must generally be preserved to achieve a functionally-correct translation for an application. A key feature of DIF in this regard is the `actor` block, which is introduced in Section 3.9.

To illustrate actor specification, we take the FFT operations in Ptolemy II and in the Autocoding Toolset as examples. In Ptolemy II, actors are implemented in Java and invoked through a Java classpath. The FFT actor in Ptolemy II is thus referred to as *ptolemy.domains.sdf.lib.FFT*. In the Autocoding Toolset, actors are called *domain primitives*, and each domain primitive is referred to by its library identifier. The FFT domain primitive in the Autocoding Toolset is referred to as *D_FFT*.

In exporting Ptolemy II to DIF, an actor's parameters and *IOPort-Relation* connections are specified as actor attributes. The built-in actor attributes `PARAMETER`, `INPUT`, and `OUTPUT` in DIF indicate the parameters and interface connections of an actor. A complete DIF actor block for the Ptolemy FFT actor is presented in Figure 26. The Ptolemy FFT actor has a parameter *order* and two IOPorts, *input* and *output*. Therefore, in the corresponding DIF actor specification, attribute *order* (with attribute type `PARAMETER`) specifies the FFT order. In addition, attributes *input* (with attribute type `INPUT`) and *output* (with attribute type `OUTPUT`) specify the incoming edge and outgoing edge that connect to the corresponding Ptolemy II IOPorts.

In the Autocoding Toolset, input/output connections and function configuration parameters of a domain primitive are all viewed as parameters. In the D_FFT domain primitive, parameter *X* specifies its input, parameter *Y* specifies its output, and parameter *N* specifies its length. In this case, the components of actor-specific information are all of the same tool-specific class (parameter), so the *attributeType* field in the DIF specification can simply be ignored. There is no loss of

```
actor nodeID {
   computation = "ptolemy.domains.sdf.lib.FFT;
   order : PARAMETER = integerValue or integerParameterID;
   input : INPUT = incomingEdgeID;
   output : OUTPUT = outgoingEdgeID;
}
```

Figure 26. The DIF actor specification for the Ptolemy FFT actor.

of information in leaving out the actor attribute type in such a case. A DIF specification for the D_FFT domain primitive is presented in Figure 27.

6.3 The Fork Actor

The `fork` actor is introduced in DIF as a special built-in actor. It has exactly one incoming edge and multiple outgoing edges. Conceptually, when firing, the fork actor consumes a token from its incoming edge and duplicates the same token on each of its outgoing edges. We say "conceptually" here because in an actual implementation of the fork actor, it may be desirable to achieve the same effect through careful arrangement and manipulation of the relevant buffers. The fork actor and related constructions are widely used in dataflow. For example, the fork actor can be used if a stream of data tokens is required to be "broadcast" to multiple destinations. The built-in DIF computation associated with the fork actor is called `dif.fork`.

In dataflow theory, an edge is a data path from a source node to a sink node. A dataflow edge cannot be associated with multiple sink nodes. In contrast, a *Relation* in Ptolemy II can have multiple destinations. In order to export Ptolemy II graphical representations to DIF representations, the graph mapping algorithm must take care of this structural difference. By using a `fork` actor, an edge connecting to the input of the `fork` actor, and multiple edges connecting from the `fork` actor to all relevant sink nodes, we can represent a Ptolemy II *Relation* and preserve its dataflow semantics while using the pure dataflow representations of DIF.

Figure 28 illustrates how the `fork` actor together with actor specification addresses the issues discussed above pertaining to *Relations* and *multiports* in Ptolemy II.

6.4 Exporting and Importing Tools

In order to provide a front-end for a dataflow-based design tool to cooperate with DIF and to use the DIF package, automating the exporting and importing processes for the design tool is a key important feature. Figure 25 illustrates a structured approach for this kind of automation. First, a dataflow graph mapping algorithm must be properly designed for the specific design tool T that is being interfaced to DIF. Then a DIF exporter is implemented for design tool T based on the graph mapping algorithm. It must be able to convert the graphical representation format of T to a corresponding DIF representation. Actor specification is also required during the conversion process to preserve the full functionality of actors in T. By applying the DIF front-end, the DIF exporter can translate the DIF representation to a corresponding DIF specification and complete the exporting process.

Similarly, by using the DIF front-end within the DIF package, a DIF language specification associated with an application for T can be converted automatically into an associated DIF representation. Then, based on a "reverse graph mapping algorithm," and the provided actor specifica-

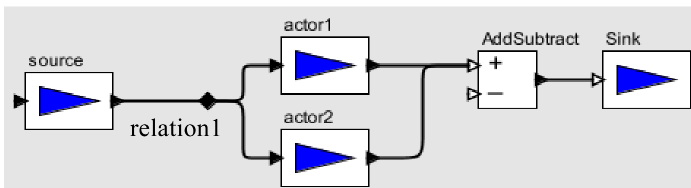
```
actor nodeID {
   computation = "D_FFT";
   N = integerValue or integerParameterID;
   X = incomingEdgeID;
   Y = outgoingEdgeID;
}
```

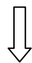
Figure 27. A DIF actor specification for the D_FFT domain primitive.

tion information, the DIF importer is able to construct the graphical representation for *T* while preserving the same functionality of the original DSP application.

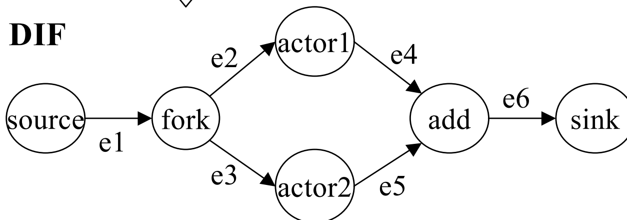
As a concrete demonstration of this approach, the DIF exporter and DIF importer for Ptolemy II are implemented according to it. With these software components, a DSP application in Ptolemy II can be exported to a DIF specification and then imported back to a Ptolemy MoML specification with all functionality preserved. Such an equivalent result from round-trip translation helps to validate the general methodology supported by DIF for dataflow graph mapping and actor specification.

Ptolemy II





DIF




```
dif graph1 {
  topology {
    nodes = source, fork, actor1, actor2, add, sink;
    edges = e1 (source, fork), e2 (fork, actor1), e3 (fork, actor2),
            e4 (actor1, add), e5 (actor2, add), e6 (add, sink);
  }
  actor fork {computation = "dif.fork";}
  actor add {
    computation = "dif.actor.lib.AddSubtract";
    plus : INPUT = e4, e5;
    output : OUTPUT = e6;
  }
}
```

Figure 28. Mapping the Ptolemy II graphical representation to a DIF representation and the corresponding DIF specification.

7 Porting DSP Applications

DIF is proposed to be a standard language for specifying dataflow graphs in all well-defined dataflow models. To this end, users are able to transfer information associated with DSP applications across different dataflow-based design tools. The objective of this porting mechanism is to provide, with a high degree of automation, a solution such that an application constructed in one design tool can be ported to another design tool with enough details preserved throughout the translation to ensure executability on the associated set of target embedded processing platforms. Because different design tools support different sets of underlying embedded processing platforms, porting DSP applications across design tools is effectively equivalent to porting them across those underlying platforms. Thus, the proposed DIF porting mechanism not only facilitates technology transfer at the level of application models, but also provides portability across target platforms.

In this section, we introduce the porting mechanism in detail. In the next section, we demonstrate that this mechanism is a feasible solution through an example of a synthetic aperture radar (SAR) benchmark application that is transferred between the MCCI Autocoding Toolset and Ptolemy II. These tools are significantly different in nature and the ability to automatically port an important application such as SAR across them is a useful demonstration of the DIF porting mechanism.

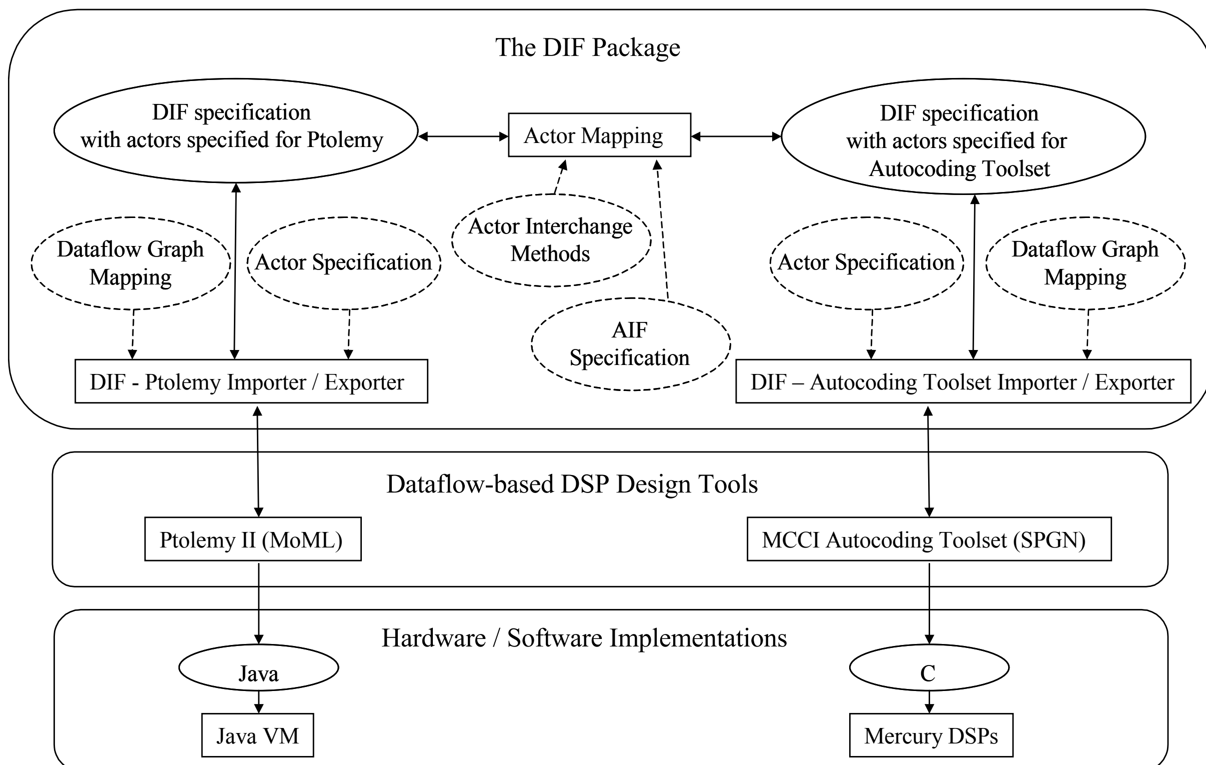


Figure 29. The DIF Porting Mechanism.

7.1 The DIF Porting Mechanism

Figure 29 illustrates the structured porting mechanism supported by DIF. This approach consists of three major steps: *exporting*, *actor mapping*, and *importing*. We take porting from the Autocoding Toolset to Ptolemy II as an example to introduce the porting mechanism in detail.

In this example, the first step is to export a DSP application developed in the Autocoding Toolset to the corresponding DIF specification. In this stage, the actor information (actor specifications in the DIF `actor` block) is specified for the Autocoding Toolset. With the DIF-Autocoding Toolset exporter/importer pair, this exporting process can be done automatically. The second step invokes the *actor mapping* mechanism to map DSP computational modules from Autocoding Toolset domain primitives to Ptolemy II actors. In other words, the actor mapping mechanism interchanges the tool-dependent actor information in the DIF specification. The final step is to import the DIF specification with actor information specified for Ptolemy II to the corresponding Ptolemy II graphical representation and then from the graphical representation to an equivalent Ptolemy II MoML specification. This importing process is handled by the DIF-Ptolemy exporter/importer automatically.

The key advantage of using a DIF specification as an intermediate state in achieving such efficient porting of DSP applications is the comprehensive representation in the DIF language of functional semantics and component/subsystem properties that are relevant to design and implementation of DSP applications using dataflow graphs. Except for the `actor` block, a DIF specification for a DSP application represents the same semantic information regardless of which design tool is importing it. Such unique semantic information is an important basis for our porting mechanism, and porting DSP applications can be achieved by properly mapping the tool-dependent actor information while transferring the dataflow semantics unaltered. Actor mapping thus plays a critical role in the porting process, and the following sub-sections describe the actor mapping process in more detail.

7.2 Actor Mapping

The objective of *actor mapping* is to map an actor in a design tool to an actor or to a set of actors in another design tool while preserving the same functionality. Because different design tools usually provide different sets of actor libraries, problems may arise due to *actor absence*, *actor mismatch*, and *actor attribute mismatch*.

If a design tool does not provide the corresponding actor in its library, we encounter the *actor absence* problem. For example, at the time when we carried out the experiments described here, Ptolemy II did not provide a matrix transpose computation but the Autocoding Toolset did. If corresponding actors exist in both libraries, but functionalities of those actors do not completely match, we have an instance of the *actor mismatch* problem. For example, the FFT domain primitive in the Autocoding Toolset allows designers to select the range of the output sequence, but the FFT actor in Ptolemy II does not provide this function. *Actor attribute mismatch* arises when attributes are mapped between actors but the values of corresponding attributes cannot be directly interchanged. For example, the parameter *order* of the Ptolemy II FFT actor specifies the FFT order, but the corresponding parameter N of the Autocoding Toolset FFT domain primitive specifies the length of FFT. As a result, in order to correctly map between *order* and N, the equation

$$.N = 2^{order}$$

must be satisfied.

The actor interchange format can significantly ease the burden of actor mismatch problems by allowing a designer a convenient means for making a one-time specification of how multiple modeling components in the target design tool can provide a subgraph such that the subgraph functionality is compatible with the actor in the source tool that is associated with a given actor mismatch problem. In addition to providing automation in the porting process, such conversions reduce the need for users to introduce new actor definitions in the target model, thereby reducing user effort and code bloat. Similarly, actor interchange methods can solve attribute mismatch problems by evaluating a target attribute in a consistent, centrally-specified manner, based on any subset of source attribute values. For absent actors, most design tools provide ways to create actors through some sort of actor definition language. Once users determine equivalent counterparts for absent and mismatched actors, our actor mapping mechanism can take over the job cleanly and efficiently.

Figure 29 illustrates our actor mapping approach to the porting mechanism.

7.3 The Actor Interchange Format

Actor information associated with a DSP application is described in the DIF `actor` block by specifying a built-in *computation* attribute and other actor attributes associated with the built-in attribute types *PARAMETER*, *INPUT*, and *OUTPUT*. Specifying actor information in the DIF actor block is referred to as *actor specification*. In order to map actor information from a source design tool to a target design tool, the actor mapping mechanism must be able to modify actor attributes and their values in DIF specifications. How to carry out this mapping process is generally based on the provided (input) actor interchange information.

The *actor interchange format* (AIF) is a specification format dedicated to specifying actor interchange information within DIF specifications. The AIF syntax consists of the actor-to-actor mapping block and the actor-to-subgraph mapping block. The actor-to-actor mapping block specifies the mapping information of computations and actor attributes from a source actor (an actor in the source design tool) to a target actor (an actor in the target design tool). On the other hand, the actor-to-subgraph mapping block specifies the mapping from a source actor to a subgraph consisting of a set of actors in the target design tool and depicts the topology and interface of this subgraph. The actor-to-subgraph mapping block is designed for use when a matching standalone actor in the target tool is unavailable, inefficient or otherwise undesirable to use in the context at hand. Subsections 7.3.1 and 7.3.2 introduce the syntax of AIF .

7.3.1 The Actor-to-Actor Mapping Block

Figure 30 illustrates the actor-to-actor mapping block. In the first line, the keyword `actor` indicates the start of an actor-to-actor mapping block. The *srcActor* and *trgActor* specifiers designate the computations (built-in *computation* attribute) of the source actor and target actor, respectively. A method *methodID* is given optionally to specify a prior condition for this mapping (i.e., a condition that must be satisfied in order to trigger the mapping). The square braces around the *methodID* field in Figure 30 indicate that this field is optional. Arguments *arg1* through *argN* can be assigned values or expressions of source actor attributes. At runtime, this method can determine whether or not the mapping should be performed based on the values of the selected source attributes.

The AIF provides four ways to specify or map to the target attribute values, each of which corresponds to a statement in the above syntax. First, it allows users to directly assign a value *value* for a target attribute *trgAtID*. The supported value types are those introduced in section 3.10. Second, a target attribute *trgAtID* can be mapped from the corresponding source attribute *srcAtID*. If *methodID* is not given in this statement, the value of *trgAtID* is directly assigned by the value of *srcAtID*. On the other hand, a method *methodID* can optionally be given to evaluate or conditionally assign the value of *trgAtID* based on the runtime values of source actor attributes. Finally, the AIF also provides syntax for one-to-multiple attribute mapping and multiple-to-one attribute mapping. For such purposes, a list of identifiers can be used as an attribute value. Note that every actor attribute can have an optionally specified *type* associated with it. For related details, see the DIF attribute blocks and DIF actor block in Section 3.7 and Section 3.9.

7.3.2 Actor-to-Subgraph Mapping Block

Figure 31 illustrates the actor-to-subgraph mapping block. The keyword *graph* in this context indicates the start of an actor-to-subgraph mapping block. The *trgGraph* term specifies the identifier or computation in order to invoke a component representing a subgraph in the target design tool and *srcActor* specifies the computation of the source actor. As with the actor-to-actor mapping block, a method *methodID* and its arguments can be optionally given to determine whether a triggering condition is satisfied.

The *topology* block is used to portray the topology of *trgGraph* and the *interface* block defines the interface ports of *trgGraph*. The AIF syntax for the topology and interface blocks is the same as that for the corresponding blocks in the DIF language. Moreover, the AIF allows users to spec-

```
actor trgActor <- srcActor | [methodID(arg1, ..., argN)] {
   trgAtID : type = value;
   trgAtID : type <- srcAtID : type | methodID(arg1, ..., argN);
   trgAtID1 : type, ..., trgAtIDn : type <- srcAtID : type;
   trgAtID : type <- srcAtID1 : type, ..., srcAtIDn : type;
}
```

Figure 30. Illustration of the actor-to-actor mapping block.

ify mappings from the interface attributes, *srcAtID* with built-in type INPUT or OUTPUT, of the source actor to the interface ports of the *trgGraph*.

The actor information of every node in *trgGraph* is specified in each `actor` block. The syntax of the AIF actor block is almost the same as the DIF `actor` block. In addition, the AIF provides syntax to map the source actor attribute *srcAtID* to the target attribute *trgAtID* while optionally taking a method for evaluating or conditionally assigning the attribute value. Moreover, multiple-to-one attribute mapping is also supported.

7.4 Actor Interchange Methods

The methods optionally specified in the actor-to-actor mapping block and actor-to-subgraph mapping block are used to perform conditional checks or to evaluate attribute values. They are referred to as *actor interchange methods*. A set of commonly-used actor interchange methods are defined in a built-in Java class in the DIF package. Users can extend this class and design more specific interchange methods for more complicated or specialized actor mapping scenarios. Every method used in an AIF specification must be defined in this built-in class or in one of the classes derived from it. Based on the explicit classpath and the method's signature, the correct method is invoked through Java reflection.

```
graph trgGraph <- srcActor | [methodID(arg1, ..., argN)] {
   topology {
      nodes = nodeID, ..., nodeID;
      edges = edgeID (sourceNodeID, sinkNodeID),
              ...,
              edgeID (sourceNodeID, sinkNodeID);
   }
   interface {
      inputs = portID : nodeID <- srcAtID : INPUT,
              ...,
              portID : nodeID <- srcAtID : INPUT;
      outputs = portID : nodeID <- srcAtID : OUTPUT,
              ...,
              portID : nodeID <- srcAtID : OUTPUT;
   }
   actor nodeID {
      computation = "stringDescription";
      trgAtID : type = value;
      trgAtID : type = ID;
      trgAtID : type = ID1, …, IDn;
      trgAtID : type <- srcAtID : type | methodID(arg1, ..., argN) ];
      trgAtID : type <- srcAtID1 : type, ..., srcAtIDn : type;
   }
}
```

Figure 31. Illustration of the actor-to-subgraph mapping block.

In the DIF package, the following actor interchange methods are built-in:

- `ifExpression("expression")`: this method evaluates the given Boolean expression and returns true or false based on the evaluation;
- `assign("expression")`: this method evaluates the given expression and returns the result of the evaluation;
- `conditionalAssign("valueExpression", "conditionalExpression")`: this method returns the value of `valueExpression` if `conditionalExpression` is true, and throws an exception otherwise.

Note that the attributes of the source actor can be used as variables in expressions and their values are used at runtime during evaluation. How to evaluate expressions is also an important issue in actor mapping.

Ptolemy II provides an efficient Java package, `ptolemy.data.expr`, for representing variables as well as parsing and evaluating expressions; we have employed this package in the implementation of AIF.

7.5 An Actor Interchange Specification Example: FFT

Although the Autocoding Toolset and Ptolemy II both provide FFT operations, actor mismatch and attribute mismatch problems still exist between the two versions. The Autocoding Toolset FFT domain primitive involves a parameter X for data input, parameter Y for data output, parameter N for FFT length, and parameter $F1$ for selecting between an FFT or IFFT (inverse FFT) operation. On the other hand, the Ptolemy II FFT actor has parameter *order*, input IOPort *input*, and output IOPort *output*. Clearly, an actor mismatch problem arises because the FFT domain primitive provides both FFT and IFFT operations, but the Ptolemy FFT actor does not. In this case, the Autocoding Toolset FFT domain primitive can be mapped to the Ptolemy FFT actor only when its parameter *FI* is not set to indicate IFFT. Moreover, an attribute mismatch problem arises because the FFT domain primitive uses the FFT length but the Ptolemy FFT actor uses the FFT order. Therefore, the parameter N can be mapped to the parameter *order* only when the condition

$$N = 2^{order}$$

is satisfied. Here, *N* and *order* are integers. An actor interchange specification for mapping the FFT operation from the Autocoding Toolset to Ptolemy II is presented in Figure 32.

The library identifier of the Autocoding Toolset FFT domain primitive is D_FFT. The classpath of the Ptolemy FFT actor is `ptolemy.domains.sdf.lib.FFT`. D_FFT can be mapped to `ptolemy.domains.sdf.lib.FFT` if the actor interchange method *ifExpression* evaluates $(F1 == 0)$ and returns *true*. The parameter *order* of the Ptolemy FFT actor is assigned to

```
actor ptolemy.domains.sdf.lib.FFT <- D_FFT | ifExpression("FI == 0") {
  order : PARAMETER <- N | conditionalAssign(
      "log(N)/log(2)","(log(N)/log(2)) - rint(log(N)/log(2)) == 0");
  input : INPUT <- X;
  output : OUTPUT <- Y;
}
```

Figure 32. The actor interchange specification of mapping the FFT operation.

$(log(N)/log(2))$ if $(log(N)/log(2))$ is an integer. Therefore, the actor interchange method *conditionalAssign* evaluates $(log(N)/log(2))$ and returns the result if

$$log(N)/log(2) - rint(log(N)/log(2)) == 0 \quad (1)$$

is true, where `rint` is a function that rounds its argument to the nearest integer. Note that if the expression in (1) evaluates to false, then *conditionalAssign* will throw an exception indicating that the attribute mapping fails.

Next, the value of parameter X is directly assigned to IOPort *input* for specifying the incoming edge. Similarly, the value of parameter Y is directly assigned to IOPort *output*.

The Autocoding Toolset FFT domain primitive also has a parameter B, which specifies the first point of its output sequence and a parameter M, which specifies the number of output points. The ability to select the range of the output sequence causes another actor mismatch problem because the Ptolemy II FFT actor does not support this function. Furthermore, there is a factor of N difference between the Autocoding Toolset FFT domain primitive configuration that performs the IFFT operation and the Ptolemy II IFFT actor. One way to solve this problem is to create a new FFT actor in Ptolemy, but the is time-consuming, and results in more library code than necessary to maintain. The AIF actor-to-subgraph mapping block can be used instead to solve such actor mismatch problems by combining multiple actors in the target design tool in strategic ways to construct a subgraph such that the functionality of the subgraph is compatible with the source actor.

The actor interchange specification in Figure 33 illustrates how to map a D_FFT domain primitive with the IFFT operation and selective output length to a Ptolemy II subgraph. If a D_FFT domain primitive outputs only part of its sequence — i.e., if the value of parameter N is not equal to that value of parameter M — then other Ptolemy II actors are involved to extract part of the output sequence of the FFT or IFFT actors. As a result, when $(F1 == 1)$ *&&* $(M != N)$ is true, a D_FFT domain primitive should be mapped to a Ptolemy subgraph capable of performing an IFFT operation and post-processing the output sequence. A subgraph in Ptolemy is represented by a supernode and is instantiated through the class `ptolemy.actor.TypedCompositeActor`.

The mapped subgraph consists of an IFFT actor, a Scale actor, a SequenceToArray actor, an ArrayExtract actor, and an ArrayToSequence actor connected in this order. The IFFT actor performs an IFFT operation, the Scale actor adjusts each sample by a factor of N, and the other three actors are used to extract a certain part of the output sequence. The subgraph has an input port *in* mapped from parameter X of D_FFT and an output port *out* mapped from parameter Y of D_FFT.

The classpaths of IFFT, SequenceToArray, ArrayExtract, and ArrayToSequence are specified in the *computation* attributes. Moreover, the parameter *order* of IFFT is mapped from D_FFT parameter N and its value is assigned $(log(N)/log(2))$ if $(log(N)/log(2))$ is an integer. The parameter *factor* of Scale is mapped from N. Then, SequenceToArray converts *arrayLength* samples to an array and its parameter *arrayLength* is mapped from N. Next, ArrayExtract extracts *extractLength* elements starting from *sourcePosition* in the input array and puts them into an output array with length *outputArrayLength* starting from *destinationPosition*. Its parameter *sourcePosition* is mapped from D_FFT parameter B. Another attribute mismatch problem arises because the array starting index in Ptolemy II is 0 but it is 1 in the Autocoding Toolset. The actor interchange method *assign* solves the problem by returning $(B-1)$. Finally, ArrayToSequence converts an array to *arrayLength* samples and *arrayLength* is mapped from D_FFT parameter M.

```
graph ptolemy.actor.TypedCompositeActor <- D_FFT
     | ifExpression("FI == 1 && M != N") {
  topology {
    nodes = IFFT, Scale, SequenceToArray, ArrayExtract, ArrayToSequence;
    edges = e1 (IFFT, Scale), e2 (Scale, SequenceToArray),
            e3 (SequenceToArray, ArrayExtract),
            e4 (ArrayExtract, ArrayToSequence);
  }
  interface {
    inputs = in : IFFT <- X;
    outputs = out : ArrayToSequence <- Y;
  }
  actor IFFT {
    computation = "ptolemy.domains.sdf.lib.IFFT";
    order : PARAMETER <- N | conditionalAssign(
       "log(N)/log(2)", "(log(N)/log(2))-rint(log(N)/log(2)) == 0");
    input : INPUT = in;
    output : OUTPUT = e1;
  }
  actor Scale {
    computation = "ptolemy.actor.lib.Scale";
    input : INPUT = e1;
    output : OUTPUT = e2;
    factor : PARAMETER <- N;
  }
  actor SequenceToArray {
  computation = "ptolemy.domains.sdf.lib.SequenceToArray";
  input : INPUT = e2;
  output : OUTPUT = e3;
  arrayLength : PARAMETER <- N;
  }
  actor ArrayExtract {
    computation = "ptolemy.actor.lib.ArrayExtract";
    input : INPUT = e3;
    output : OUTPUT = e4;
    sourcePosition : PARAMETER <- B | assign("B-1");
    extractLength : PARAMETER <- M;
    destinationPosition : PARAMETER = 0;
    outputArrayLength : PARAMETER <- M;
  }
  actor ArrayToSequence {
    computation = "ptolemy.domains.sdf.lib.ArrayToSequence";
    input : INPUT = e4;
    output : OUTPUT = out;
    arrayLength : PARAMETER <- M;
  }
}
```

Figure 33. An actor interchange specification of actor-to-subgraph mapping of IFFT operation.

7.6 Summary

By supporting automatic exporting and importing for source and target design tools, the first and third porting steps are achieved. With the actor interchange format and actor interchange methods for actor mapping, the entire three-step DIF porting mechanism is demonstrated, as illustrated in Figure 29.

The actor interchange methods and the corresponding AIF syntax are able to solve most attribute mismatch problems because the target attribute value can be expressed conditionally based on all source attribute values and users can design actor interchange methods for different scenarios. In addition, actor-to-subgraph mapping can solve actor mismatch problems because users can collect a group of target actors to construct a subgraph such that the functionality of the subgraph is compatible with that of the source actor. If an actor is absent, manually creating the corresponding actor is the last resort; the features in AIF greatly help to minimize the need for doing this. Once users make suitable provisions for all of the absent actors, the actor mapping mechanism associated with AIF can take over the job in an efficient, systematic fashion.

DIF is capable of porting DSP applications across dataflow-based design tools without any standard library. In this case, the actor interchange format acts as a standard specification format to specify the interchange information between tools. However, cooperating with a standard library for providing functional interfaces for actors can further facilitate the porting process. Even with a standard library, however, the actor interchange format is still essential in the DIF package porting methodology — the AIF is required for mapping actors between different design tools and the standard library.

8 Final Words

This document has introduced the DIF language, the DIF package, and the supported dataflow models in DIF. We have described a DIF-based methodology to automate the exporting and importing processes. Finally, we developed the DIF porting mechanism, which streamlines the process of transferring dataflow-based designs across different design tools.

As DIF is actively being researched and further developed, we welcome any questions or suggestions regarding the language, the underlying set of formal models, or directions for future work.

9 Acknowledgements

The primary sponsors of the Dataflow Interchange Format Project have been the **U.S. Defense Advanced Research Projects Agency**; **Management, Communications & Control, Inc.**; and the **Semiconductor Research Corporation**.

We are also grateful to the the following people who have made significant contributions to the the DIF project, and to the work that led up to it in the Maryland DSPCAD Research Group: Celine Badr, Neal Bambha, Bishnupriya Bhattacharya, Nitin Chandrachoodan, Carl Ecklund, Ruirui Gu, Fiorella Haim, Fuat Keceli, Hojin Kee, Mukul Khandelia, Vida Kianzad, Dong-Ik Ko, Sumit Lohani, Sebastian Puthenpurayil, Christopher Robbins, Sankalita Saha, Perttu Salmela, Shahrooz Shahparnia, Mainak Sen, Chung-Ching Shen, Gary Spivey, and Ankush Varma.

10 Document Version

The date of the initial version of this document is June 16, 2007.

The date of this revision, which incorporates minor fixes and clarifications with respect to the initial version, is June 16, 2007.

References

[1] H. Andrade and S. Kovner. Software synthesis from dataflow models for embedded software design in the G programming language and the LabVIEW development environment. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1998.

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

[3] S. S. Bhattacharyya. Hardware/software co-synthesis of DSP systems. In Y. H. Hu, editor, *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, pages 333-378. Marcel Dekker, Inc., 2002.

[4] 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.

[5] 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.

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

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

[8] 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, CA 94720, 1993.

[9] E. F. Deprettere, T. Stefanov, S. S. Bhattacharyya, and M. Sen. Affine nested loop programs and their binary cyclo-static dataflow counterparts. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pages 186-190, Steamboat Springs, Colorado, September 2006.

[10] J. Eker and J. W. Janneck. CAL language report, language version 1.0 — document edition 1. Technical Report UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley, December 2003.

[11] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity — the Ptolemy approach. *Proceedings of the IEEE*, January 2003.

[12] E. Gagnon. *SableCC, an object-oriented compiler framework*. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, March 1998.

[13] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya. DIF: An interchange format for dataflow-based design tools. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2004. To appear.

[14] F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya. *First version of a dataflow interchange format*. Technical Report UMIACS-TR-2002-98, Institute for Advanced Computer Studies, University of Maryland at College Park, November 2002. Also Computer Science Technical Report CS-TR-4418.

[15] E. A. Lee and S. Neuendorffer. MoML - a modeling markup language in XML version 0.4. Technical Report UCB/ERL M00/12, Electronics Research Laboratory, University of California at Berkeley, March 2000.

[16] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235-1245, September 1987.

[17] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cyclo-static dataflow. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1995.

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

[19] C. B. Robbins. *Autocoding Toolset software tools for automatic generation of parallel application software*. Technical report, Management, Communications & Control, Inc., 2002.

[20] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[21] R. S. Stevens. The processing graph method tool (PGMT). In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pages 263-271, July 1997.

[22] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction*, 2002.

[23] J. Teich and S. S. Bhattacharyya. Analysis of dataflow programs with interval-limited data-rates. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 507-518, Samos, Greece, July 2004.