Technical Report #UMIACS-TR-2004-66, Institute for Advanced Computer Studies,
University of Maryland at College Park, November 2004

Dataflow Interchange Format Version 0.2

Chia-Jui Hsu and Shuvra S. Bhattacharyya

Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742
{jerryhsu, ssb}@eng.umd.edu

Abstract

The Dataflow Interchange Format (DIF) is a standard language to specify mixed-grain dataflow models for digital signal, image, and video processing (DSP) systems and other streaming-related application domains. Major objectives of the DIF project are to design this standard language; to provide an extensible repository for representing, experimenting, and developing dataflow models and techniques; and to facilitate technology transfer of applications across DSP design tools. The first version of DIF [9, 13] has demonstrated significant progress towards these goals. The subsequent phase of the DIF project, which we discuss in this report, is focusing on improving the DIF language and the DIF package to represent more sophisticated dataflow semantics and exploring the capability of DIF in transferring DSP applications and technology. This exploration has resulted so far in an approach to automate exporting and importing processes and a novel solution to porting DSP applications through DIF. This report introduces the DIF language version 0.2 along with the DIF package, the supported dataflow models, the approach to exporting and importing, and the newly proposed porting mechanism.

1 Introduction

Modeling DSP applications through coarse-grain dataflow graphs is widespread in the DSP design community, and a variety of dataflow models [1, 4, 6, 7, 14] have been developed for dataflow-based design. Nowadays, a growing set of DSP design tools support such dataflow semantics [3]. 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.

The first version of DIF demonstrated partial success in achieving such goals. However, its language syntax and accompanying intermediate representations are insufficient to handle more complicated dataflow models as well as to transfer DSP applications through significant levels of automation. Therefore, in the subsequent phase of our work on DIF, version 0.2 of the DIF language was developed and the associated DIF package was also extended to address these challenges. With the enhanced DIF language and DIF package, advanced dataflow modeling techniques such as parameterized synchronous dataflow [1] and boolean-controlled dataflow [7] can be fully supported.

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 the problems in exporting and importing are design-tool-specific, practical implementation issues are quite common among different design tools. These issues have been carefully studied and we describe our approaches to addressing them in this report.

The problem of transferring DSP applications across design tools with a high degree of automation has also been investigated. 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 dataflow-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 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.

The organization of this report is as follows. In Section 2, we review the basic concepts of dataflow graphs and introduce the hierarchy structure in DIF. Then we describe the DIF language version 0.2 in Section 3. Next, we illustrate dataflow models in DIF through examples of DIF specifications in Section 4. In Section 5, we introduce the DIF package and discuss the overall methodology of design using DIF. In Section 6, we discuss critical problems in exporting and importing DIF specifications from DSP design tools and describe our approaches to these problems. In Section 7, we develop the porting mechanism of DIF and discuss associated actor mapping issues. In Section 8, we show the feasibility of the DIF porting capabilities by demonstrating the porting of a Synthetic Aperture Radar application from the MCCI Autocoding Toolset [16, 17] to Ptolemy II [10, 11, 12]. In the final section, we list some major directions for future work.

2 Dataflow Graphs and Hierarchical Dataflow Representation

2.1 Dataflow Graph

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 $src(e) \in V$, $snk(e) \in V$, and e can be denoted as $v1 \rightarrow v2$. 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) 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 a z^{-1} operator.

Dataflow graphs naturally capture the data-driven property in 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 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 logic operations to DSP subsystems such as filters or FFT algorithm. If an actor represents an indivisible operation, 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 logic 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 many sophisticated DSP applications, the mixed-grain dataflow graph of an overall system consists of several supernodes, and each top-level supernode can be further refined into another mixed-grain dataflow graph, possibly with additional (nested) supernodes.

One way to describe such complicated systems is to flatten the associated hierarchies 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 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 $s \cong G'$ and $H' = (G', I', M')$. The *mapping* M in a hierarchy $H = (G, I, M)$ is a set containing all mappings (to subhierarchies) of supernodes s in $G = (V, E)$; that is, $\forall s \in S \subset V$, $\{s \Rightarrow H'\} \in M$.

The *interface* I in hierarchy H is a set consisting of all interface ports in H. An *interface port* (or simply called *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 inside node, and conversely, an output port is an exit point for tokens moving from an inside 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 inside 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 specify hierarchical dataflow graphs in the DIF language, which is introduced 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 is 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.

DSP applications specified by 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 [8]. The complete SableCC grammar of the Dataflow Interchange Format is presented in Appendix A.

3.1 Dataflow Interchange Format Language Version 0.2

The first version of the DIF language [9], version 0.1, was the first attempt to approach aforementioned goals. In DIF version 0.1, we demonstrated the capability of conveniently specifying and manipulating fundamental dataflow models such as SDF and CSDF. Nonetheless, its semantics is insufficient to describe in detail more advanced dataflow semantics and to specify actor-specific information. As a result, the DIF language has been 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.

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

DIF version 0.2 consists of eight blocks: basedon, topology, interface, parameter, refinement, built-in attribute, user-defined attribute, and actor. Those blocks specify different aspects of dataflow semantics and modeling information. The following subsections introduce the syntax of the DIF language. Items in boldface are keywords. Non-bold words should be specified by users.

3.2 The Main Block

A dataflow graph is specified in the main block consisting of two arguments, *dataflowModel* and *graphID*, followed by the main braces. The *dataflowModel* keyword specifies the dataflow model of the graph. The *graphID* specifies the name (identifier) of the graph. The following is the overall of the main block:

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

The eight blocks are defined in the main braces. Each block starts with a block keyword and the content is enclosed by braces. Statements inside block braces end with semicolons. Conventionally, identifiers in DIF only consist of alphabetic, underscore, and digit characters. However, DIF also supports arbitrarily-composed identifiers by enclosing them between two dollar-sign characters. The basedon, topology, interface, parameter and refinement blocks should be defined in this particular order. Except the topology block, however, all other blocks are optional. A top-level graph specification does not have to define the interface block.

3.3 The Basedon Block

```
basedon { graphID; }
```

The *basedon* block provides a convenient way to refer to a pre-defined graph, which is specified by *graphID*. 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, duplicated subgraphs usually have the same topologies but different parameters or attributes. The basedon block is designed to support this characteristic and promote conciseness and code reuse.

3.4 The Topology Block

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

The *topology* block specifies the topology of a dataflow graph $G = (V, E)$. 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 for a node definition statement and node identifiers, *nodeID*s, are listed following the keyword and equal sign. Similarly, *edges* is the keyword for an edge definition statement and edge definitions are listed in a similar fashion. An edge definition, *edgeID (sourceNodeID, sinkNodeID)*, consists of three arguments in order to specify a directed edge: the edge identifier *edgeID*, the source node identifier *sourceNodeID*, and the sink node identifier *sinkNodeID*.

3.5 The Interface Block

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

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 inside association $p_i : v_i$. Similarly, an output definition statement defines every output port $p_o \in out(I)$ and the corresponding inside 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*, consists of two arguments, a port identifier and its associated node identifier. DIF permits defining an interface port without an associated node, so *assocNodeID* is optional.

3.6 The Parameter Block

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

In many DSP applications, designers often parameterize important attributes such as the frequency of a sine wave generator and the order of a FFT actor. In interval-rate, locally-static dataflow [18], unknown production and consumption rates are specified by their minimum and maximum values. In parameterized dataflow [1], production and consumption rates are even allowed to be unspecified and dynamically parameterized. The *parameter* block is designed to support parameterizing values in ways like these, and to support value ranges, and value-unspecified attributes.

In a parameter definition statement, a parameter identifier *paramID* is defined and its *value* is optionally specified. DIF supports various value types and those types are introduced in Section 3.11.

DIF also supports specifying the range of possible values for a parameter. The *range* is specified as an interval such as (1, 2), (3.4, 5.6], [7, 8.9), [-3.1E+3, +0.2e-2], or a set of discrete numbers such as {-2, 0.1, +3.6E-9, -6.9e+3}, or a combination 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}.

3.7 The refinement block

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

The *refinement* block is used to represent hierarchical graph structures. For each supernode $s \in S \subset V$ in a graph $G = (V, E)$, there should be a corresponding refinement block in the DIF specification 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 or computed at runtime) 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 blank parameters in the subgraph by using parameters defined in the current graph.

Figure 1 illustrates how to use DIF to specify hierarchical dataflow graphs. In Figure 1, 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 1.

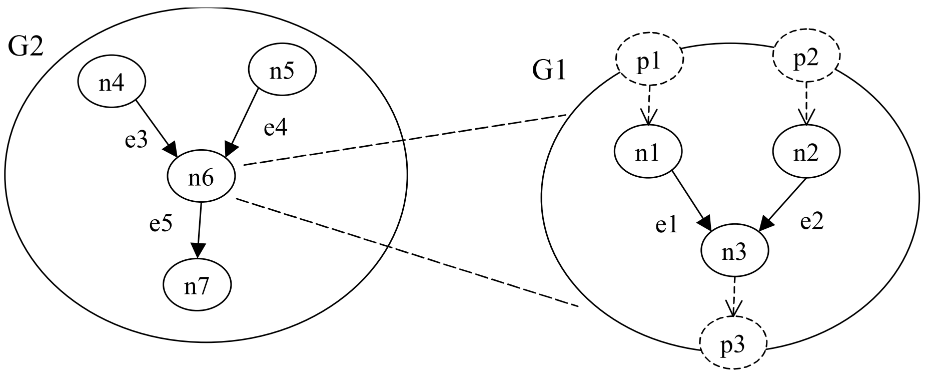
3.8 The Built-in Attribute Block

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

The keyword *builtInAttrID* points out which built-in attribute is specified. The element identifier, *elementID*, can be a node identifier, an edge identifier, or a port identifier to which the built-in attribute belongs. It can also be left blank; in this case, the built-in attribute belongs to the graph itself. DIF supports assigning attributes by a variety of value types, an identifier, or a list of identifiers. The supported value types are introduced in Section 3.11.

Usually, the built-in attribute block is used to specify dataflow modeling information. Every dataflow model in DIF can define its own built-in attributes and its own method to process those built-in attributes. The DIF language parser treats built-in attributes in a special way such that the method defined in the corresponding parser is invoked to handle them. Some dataflow models require model-specific attributes and value types, and DIF specifications for those dataflow models will be discussed in Section 4.

In general, *production*, *consumption*, and *delay* are commonly-used built-in attributes of edges in many dataflow models. For example, if *delay(e1) = 1D* and *delay(e2) = 2D*, where *D* is a delay



```
dif graph G1 {
  topology {
    nodes = n1, n2, n3;
    edges = e1 (n1, n3), e2 (n2, n3);
  }
  interface {
    inputs = p1 : n1, p2 : n2;
    outputs = p3 : n3;
  }
}
```

```
dif graph G2 {
  topology {
    nodes = n4, n5, n6, n7;
    edges = e3 (n4, n6),
            e4 (n5, n6), e5 (n6, n7);
}
refinement {
    G1 = n6;
    p1 : e3; p2 : e4; p3 : e5;
  }
}
```

Figure 1. Hierarchical graphs and the corresponding DIF specifications.

unit, the delay attribute block is specified as: `delay { e1 1; e2 2; }`. Note that the built-in attributes *production* and *consumption* are not exclusive to edges. In hierarchical dataflow models, the interface-associated node has no edge on the corresponding direction. In such cases, specifying production rates or consumption rates as port attributes is permitted in DIF.

3.9 The User-Defined Attributes Block

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

The user-defined attributes block allows designers to define and specify their own attributes. The syntax is the same as the built-in attributes block. The only difference is that this block starts with the keyword *attribute* followed by the user-defined attribute identifier, *usrDefAttrID*.

3.10 The Actor Block

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

The topology, interface, parameter, refinement and built-in attribute blocks are used to describe graph topology, hierarchical structure, and dataflow semantics. They are sufficient for applying dataflow-based analysis and optimization techniques. However, in order to preserve the functionality of DSP applications in design tools, the information supported in DIF language version 0.1 is not enough. As a result, the actor block has been created in DIF language version 0.2 to specify tool-specific actor information.

The keyword *actor* is used for the actor block. The associated *computation* is a built-in actor attribute for specifying in some way the actor's computation (what the actor does). Other actor information is specified as attributes. Explicitly, the identifiers of actor's components such as ports, arguments, or parameters are used as *attributeID* in the DIF actor block. Moreover, the type of the component can be optionally specified as *attributeType*. DIF supports three built-in actor attribute types: INPUT, OUTPUT, and PARAMETER to indicate the interface connections and parameters of an actor. Attributes can be assigned a value, or an identifier for specifying its associated element (edge, port, or parameter), or a list of identifiers for indicating multiple associated elements of the attribute.

The actor block is primarily used in exporting and importing DIF as well as porting DSP applications. Section 6 and Section 7 contain more explanations and examples of how to use the DIF actor block.

3.11 The Value Types

DIF version 0.2 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. For example, a double value can have the following formats: 123.456, +0.1, -3.6, +1.2E-3, -4.56e+7. A complex value is enclosed by parentheses as (real part, imaginary part), and the real and imaginary parts are double values. For example, a complex value 1.2E-3 - 4.56E+7 i is represented as (+1.2E-3, -4.56E+7) in DIF. Matrices are enclosed by brackets, "," is used to separate elements in a row, and ";" is used to separate rows. For example, integer matrices, double matrices, and complex matrices are expressed as [1, 2; 3, 4], [+1.2, -3.4; -0.56e+7, 7.8E-3], and [(1.0, 2.0), (3.0, 4.0); (+1.2, -3.4), (-0.56e+7, 7.8E-3)]. A string value should be double quoted as "string". A boolean value is either True or False. Finally, an array of the aforementioned value types is expressed inside braces, and all elements should be of the same type. For example, we can have an integer array as {1,2,3,4} or double array as {+0.1, -3.6, +1.2E-3, -4.56e+7}. These value types in DIF should be sufficient in most DSP applications. If a certain value type is not supported, it can be handled to some extent by representation through the string type.

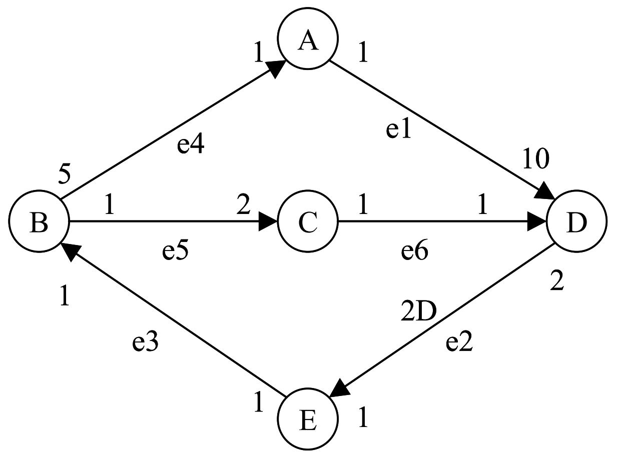
4 Dataflow Models

The DIF language is designed to specify all dataflow models for DSP and streaming related applications. In other words, its syntax and other features should be capable of describing dataflow semantics in all dataflow models of computation relevant to this class of embedded applications. DIF version 0.1 [9] has demonstrated its capability of describing CSDF, SDF, single-rate dataflow, and HSDF. DIF version 0.2 improves the feature set to support more complicated dataflow semantics, for example, Turing-complete dataflow such as BDF [7] and meta-modeling techniques such as parameterized dataflow [1]. This section reviews those dataflow models and provides examples to illustrate how to specify them in DIF.

4.1 Synchronous Dataflow

Synchronous dataflow (SDF) [4, 14] is the most popular form of dataflow modeling for DSP design. SDF permits the number of tokens produced and consumed by an actor to be a non-negative integer, which makes it very suitable for modeling *multi-rate* DSP systems. However, SDF also imposes a restriction that production and consumption rates must be fixed and known at compile-time. Therefore, an edge *e* in an SDF graph has three non-negative constant-valued attributes, *prd(e)*, *cns(e)*, and *delay(e)*. The constant restriction on production and consumption rates benefits SDF with the capability of static scheduling, optimization, and predictability [5] but at the cost 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 three edge attributes, *prd(e)*, *cns(e)*, and *delay(e)*, are specified in SDF built-in attribute blocks as *production*, *consumption*, and *delay*. Figure 2 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 2. An SDF example and the corresponding DIF specification.

4.2 Single-rate Dataflow and Homogeneous Synchronous Dataflow

In *single-rate* DSP systems, 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 equal to the number of tokens consumed on the same edge when the sink node fires. The *dataflowModel* keyword for single-rate dataflow is *SingleRate*. The single-rate dataflow model is a special case of SDF, where the production rate and consumption rate of each edge are identical. Because all nodes fire at the same average rate, DIF uses the built-in attribute *transfer* to specify token transfer rates instead of *production* and *consumption* attributes.

In *homogeneous synchronous dataflow* (HSDF), the production rate and consumption rate are restricted to be unity on all edges. HSDF is the simplest widely-used form of dataflow and 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, specifying *production* and *consumption* attributes is not necessary in HSDF.

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. Algorithms for converting between SDF, single-rate, and HSDF graphs are provided in DIF. Such conversion is illustrated in Figure 3.

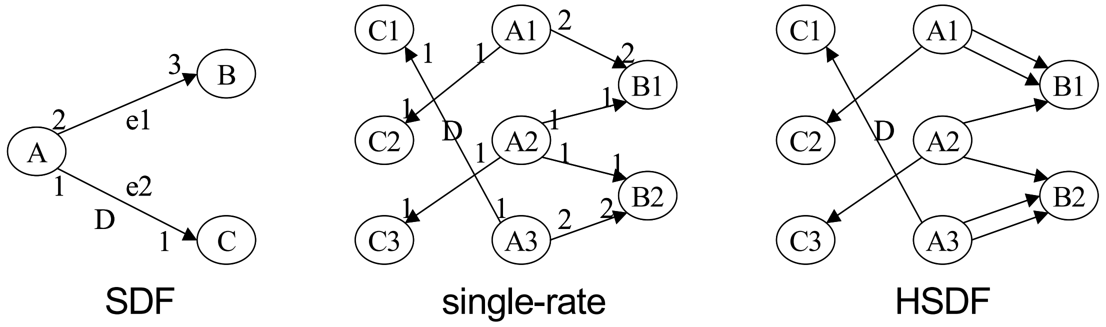
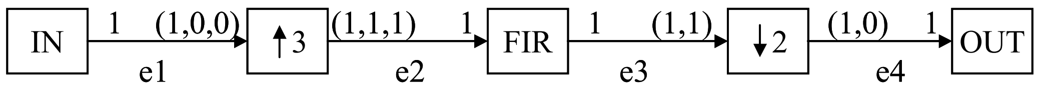


Figure 3. SDF, single-rate, and HSDF conversion.

4.3 Cyclo-static Dataflow

In *cyclo-static dataflow* (CSDF) [6], the production rate and consumption rate are allowed to vary as long as the variation forms a fixed and periodic pattern. Explicitly, 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 to 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.

The *dataflowModel* keyword for CSDF is *csdf*. Built-in attributes *production* and *consumption* are specified as 1-by-$\tau(A)$ integer matrices representing the $\tau(A)$-tuple patterns in one period. Specifically, the DIF specification for a $\tau(A)$-tuple consumption period is specified as: *consumption{ edgeID = [*$C_{e,1}$, $C_{e,2}$, ..., $C_{e,\tau(A)}$*]; }*. Figure 4 illustrates an up-sampling and down-sampling example in CSDF.



```
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 4. A CSDF example and the corresponding DIF specification.

4.4 Boolean-controlled dataflow

Boolean-controlled dataflow (BDF) [7] 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 [7]. 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 explore 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 certain "control tokens." In other words, the number of tokens that a boolean-controlled actor A produces to an edge e_o or consumes from an edge e_i during each firing is determined by *TRUE* or *FALSE* values of the control token consumed by A at that iteration, where $e_o \in out(A)$ or $e_i \in in(A)$. BDF also imposes a restriction that a boolean controlled actor can only consume one control token during each firing. The following two equations describe the boolean-controlled production and consumption rates in BDF.

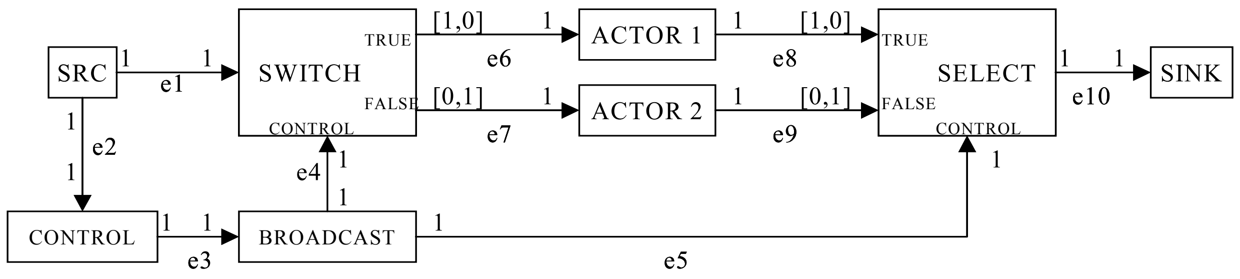
$$prd(e_o) \text{ at i-th iteration} = \begin{cases} prod\ rate1, \text{ if control token consumed by A at i-th iteration is TRUE} \\ prod\ rate2, \text{ if control token consumed by A at i-th iteration is FALSE} \end{cases}$$

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

In addition to boolean-controlled dynamic actors, other actors are required to be *regular*. A regular actor produces and consumes fixed and known numbers of tokens at compile-time; it is equivalent to an SDF actor.

The *dataflowModel* keyword for BDF is *bdf*. Built-in attributes *production* and *consumption* can be used to specify both fixed and boolean-controlled production and consumption rates. For a fixed rate, the syntax is the same as SDF; for a boolean-controlled rate, a 1-by-2 integer matrix is utilized to specify two boolean-controlled values. The first element is the rate when the control token is *TRUE*, and the second element is the rate when the control token is *FALSE*. Specifically, the following syntax shows the DIF specification for a boolean-controlled production rate.

```
production { edgeID = [trueValue, falseValue]; }
```



```
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 5. A BDF example, the corresponding pseudocode, and the DIF specification.

BDF introduces two boolean-controlled 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 a control token. The SELECT actor consumes one token from either a "true" incoming edge or a "false" incoming edge according to the value of a control token and copies that token to the outgoing edge. Figure 5 illustrates a BDF example implementing an *if-else* statement.

4.5 Parameterized Synchronous Dataflow

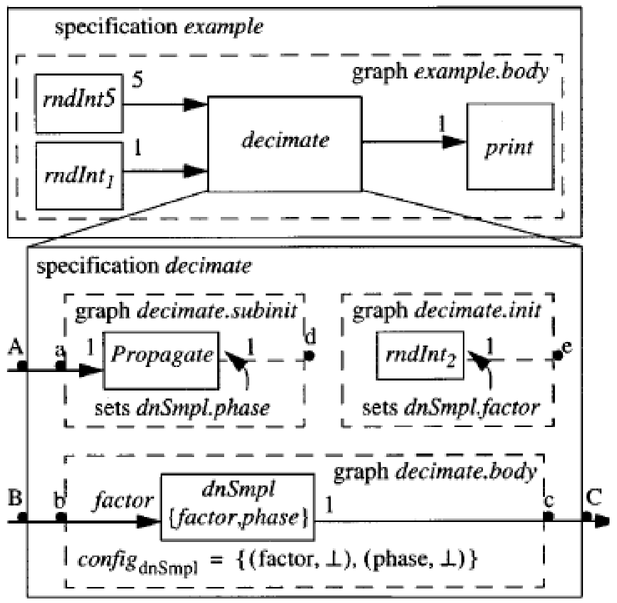
Parameterized dataflow modeling differs from other fundamental dataflow modeling techniques such as SDF, CSDF, in that it is a *meta-modeling* technique. Parameterized dataflow can be applied to any underlying "base" dataflow model that has a well-defined notion of a graph iteration. Applying parameterized dataflow in this way augments the base model with powerful capabilities for dynamic reconfiguration and quasi-static scheduling through parameterized looped schedules [1]. 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 [1].

A PSDF actor *A* is characterized by a set of parameters, *params(A)*, that can control the actor's functionality as well as the actor's dataflow behavior such as production rates and consumption rates. A configuration of a PSDF actor, $config_A$, is determined by assigning values to the parameters of A. Each parameter of an actor is either assigned a value or left unspecified. These statically unspecified parameters are assigned values at run time, thus dynamically modifying the actor's functionality.

A PSDF graph *G* is an ordered pair *(V, E)* and all statically unspecified actor parameters in *G* propagate "upwards" 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. 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 dataflow and all the parameters of Φ_i are left unspecified. 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 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(*Φ*)* and are configured by the init and subinit graphs of hierarchically higher level subsystems. This mechanism of parameter configuration is referred 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. As a result, the parent has a



```
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 6. A PSDF example and the corresponding DIF specification.

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.

Specifying such complicated meta-modeling techniques in a fully general way (so that they can operate with other models a maximally flexible way) in DIF is a challenging task. DIF separates PSDF graphs and PSDF subsystems into two modeling blocks, and the corresponding *dataflowModel* keywords for them are *psdf* and *psdfSubsystem*, respectively. Parameterization is a main feature of DIF with the parameter block and this feature is very suitable in specifying PSDF. Configurable actor attributes and non-static dataflow modeling attributes such as production rates and consumption rates are parameterized by pre-defined parameters. Unspecified parameters are defined without providing their values in the parameter block. *Upward parameters* of a PSDF subsystem can be specified in the refinement block of its supergraph. For hierarchical modeling structures in PSDF, e.g., $\Phi = subsystem(H)$, the DIF hierarchy concepts described in Section 2.2 can fully represent the associated functionality and the DIF refinement block is used to specify them.

DIF interprets a PSDF subsystem as a special intermediate graph that consists of three subgraphs, Φ_i, Φ_s, and Φ_b. In DIF specification, a PSDF subsystem cannot have the 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. The parameter reconfigurations across init, subinit, and body graphs are specified in the built-in attribute block called *paramConfig* with the following syntax.

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

Figure 6 illustrates a PSDF example in [1] and the corresponding DIF specification.

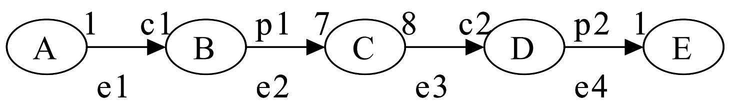
4.6 Binary Cyclo-static Dataflow

Binary CSDF (BCSDF) is a restricted form of CSDF such that the production and consumption rates are constrained to be binary vectors. In other words, elements of the BCSDF production and consumption vectors are either 0 or 1. BCSDF graphs arise naturally, for example, when converting certain process networks to dataflow and when modeling many dataflow-based hardware implementations.

The DIF specification format for BCSDF is as the same as CSDF. In some BCSDF representations, the numbers of phases can be very large. Therefore, the BCSDF intermediate representation utilizes an efficient data structure to store the production and consumption rates as bit vectors.

4.7 Interval-Rate Locally-static Dataflow

Interval-Rate Locally-static Dataflow (ILDF) [18] is proposed to analyze dataflow graphs whose component data rates are not known precisely at compile time. In ILDF graphs, the production and consumption rates remain constant throughout execution (locally-static), but only the minimum and maximum values (interval-rate) of these constants are given. DIF is capable of representing ILDF graphs by parameterizing the ILDF production and consumption rates and specifying the intervals of those parameters, which is described in Section 3.6. Figure 7 illustrates an ILDF example and the corresponding DIF specification.



```
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 7. An ILDF example and the corresponding DIF specification.

5 The DIF Package

The DIF package is a Java software package developed along with the DIF language. In general, it 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 underlying embedded processing platforms.

5.1 The DIF Representation

For each supported dataflow model, 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 the dataflow model is usually referred to as the *DIF representation*.

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 that are common to all models for manipulating graphs. 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 8 presents 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 Graph classes are used from the Ptolemy II [11] ptolemy.graph package, which is developed in collaboration between members of the DIF and Ptolemy projects, and provides data structures and methods for manipulating generic graphs. 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 8).

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

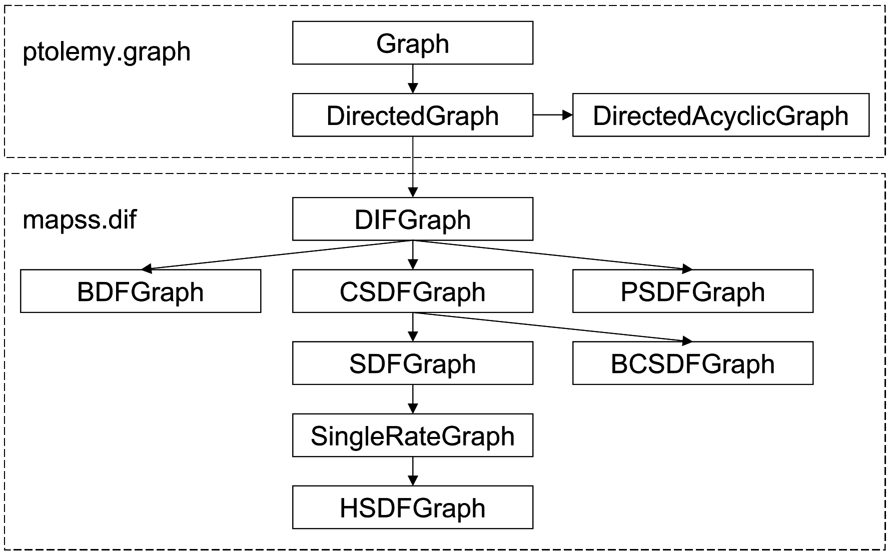


Figure 8. The class hierarchy of graphs in the DIF package.

5.2 The DIF Front-end

Although the DIF language is able to specify all dataflow models, in reality, the DIF representation is the actual format for realizing dataflow graphs and for performing analysis, scheduling, and optimization. Thus, automatic conversion between DIF specifications (.dif files) and DIF representations (graph instances) is the most 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 [8]. The flexible structure and Java integration of the SableCC compiler enables easy extensibility for parsing different dataflow graph types.

Figure 9 illustrates how the DIF front-end constructs the corresponding DIF representation (graph class) according to the 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 10 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.

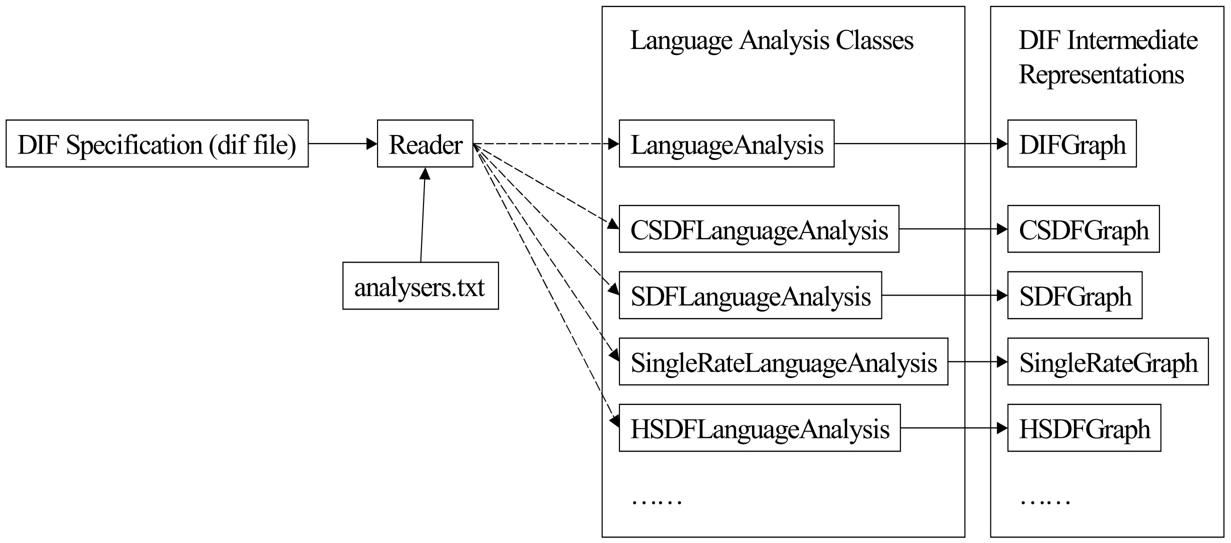


Figure 9. The DIF Front-end: from DIF specification to DIF representation.

In the DIF package, the language analysis classes (language parsers) are used for parsing the DIF language. The only differences between them are in processing the built-in attributes and in initiating the corresponding graph class. Similarly, the graph writer classes are used for writing out the dataflow semantics and they are different in handling built-in attributes. Therefore, all specialized dataflow language analysis classes are extended from the LanguageAnalysis class that constructs the most general DIFGraph. Likewise, all specialized graph writer classes are extended from the DIFWriter class, which writes out the dataflow semantics of DIFGraph instances. The extended classes are required to override only a small set of model-specific methods.

5.3 Dataflow-based Analysis, Scheduling, and Optimization Algorithms

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. Algorithms currently available in the DIF package are based primarily on well-developed algorithms such as iteration period computation, consistency validation, buffer minimization, and loop scheduling. By building on the DIF representations and existing algorithm implementations, and invoking the built-in algorithms as

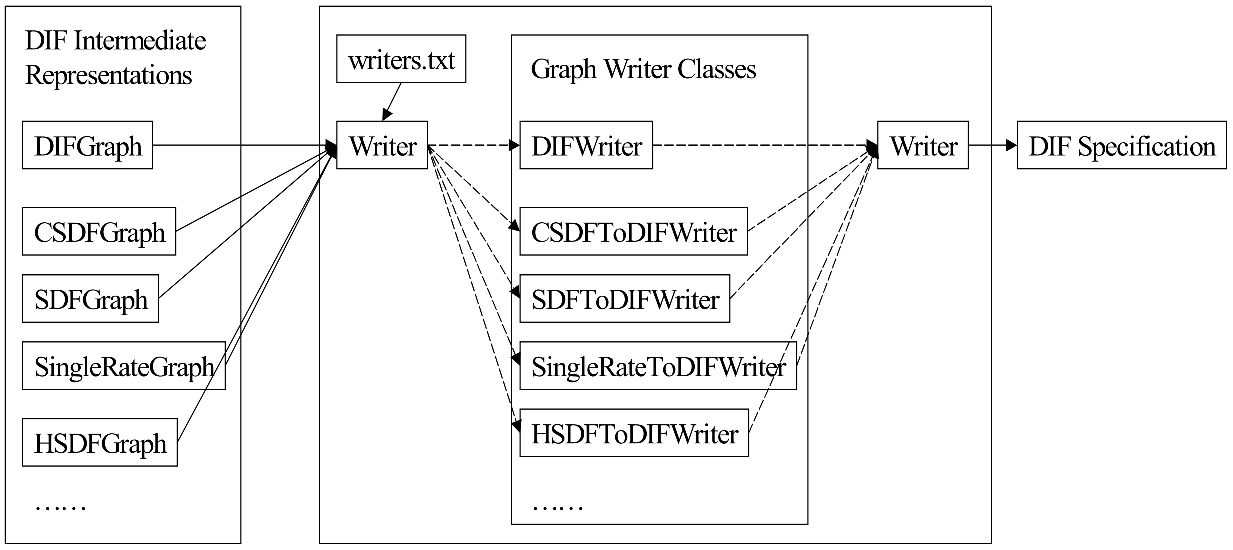


Figure 10. The DIF front-end: from DIF representation to DIF specification.

needed, emerging techniques and other new algorithm implementations can conveniently be developed and implemented in the DIF package.

The dataflow-based algorithms in the DIF package provide designers an efficient interface to analyze and optimize DSP applications. It is also worthwhile to integrate DSP design tools with the DIF package and then utilize the powerful scheduling and optimization features of the DIF package.

5.4 Methodology of using DIF

Figure 11 illustrates the conceptual architecture of DIF and the relationships among abstract dataflow models, dataflow-based DSP design tools, DIF specifications, and the DIF package. First of all, the dataflow model block in this diagram presents 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 for dataflow-based design tools represents 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 12 illustrates the implementation and end-user viewpoints of the DIF architecture. DIF supports as the core a layered design methodology covering dataflow models, the DIF language and DIF specifications, the DIF package, dataflow-based DSP design tools, and the underlying hardware and software platforms targeted by these tools.

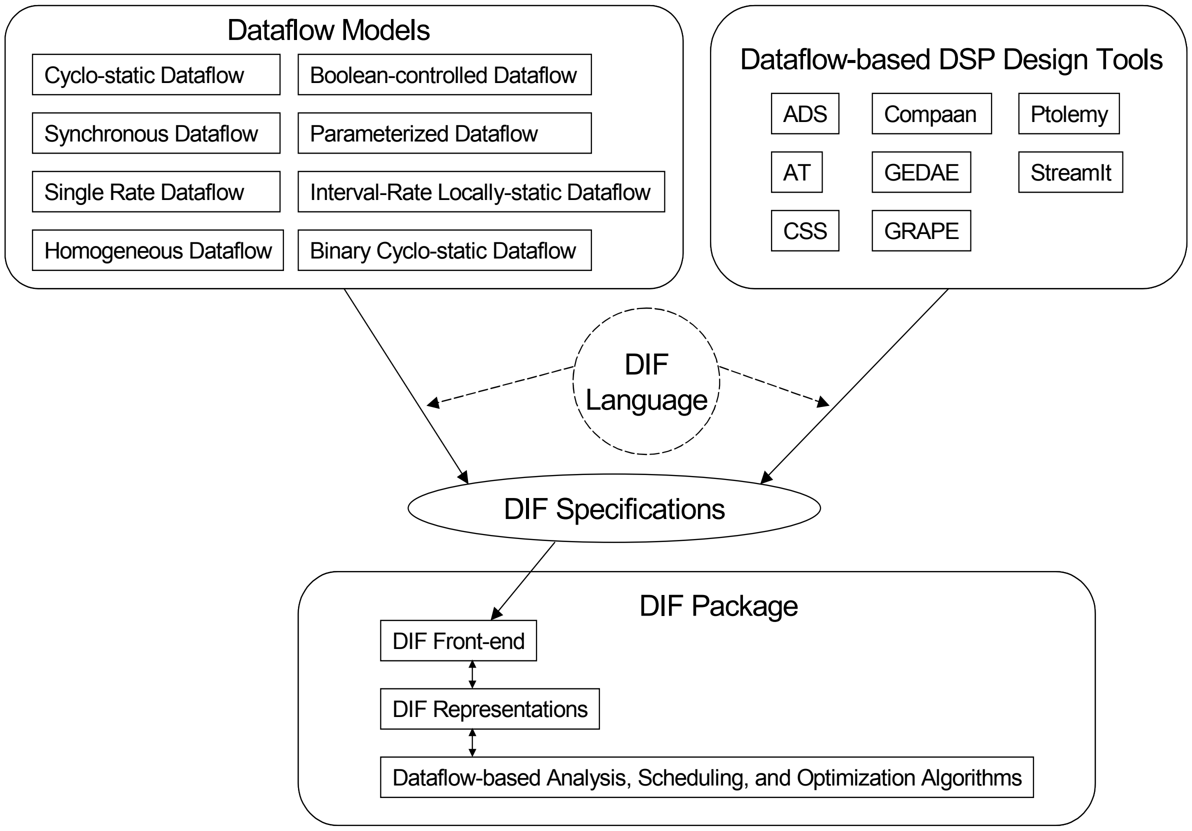


Figure 11. The relationships among dataflow models, design tools, the DIF language, DIF specifications, and the DIF package.

The dataflow models layer 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 dataflow modeling techniques can be specified in a streamlined manner as specialized DIF specifications.

The primary dataflow-based DSP design tools that we have been experimenting with in our development of DIF so far are the SDF domain of Ptolemy II, 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 are 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 a layer in our proposed DIF-based design methodology. Ptolemy II is a Java-based design environment and utilizes the Modeling Markup Language (MoML) as its textual format for specification and interchange. Ptolemy II provides multiple models of computation and a large set of libraries consisting of actors for various application domains. On the other hand, the MCCI Autocoding Toolset is based on the Processing Graph Method (PGM) semantics 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 hardware / software embedded systems layer gives examples of current embedded processing platforms supported by Ptolemy II and the Autocoding Toolset, and this layer generally represents all embedded platforms that are supported by dataflow-based DSP design tools. Ptolemy II can generate executable Java code running on the Java VM. On the other hand, the Autocoding toolset is able to generate executable C code for Mercury DSPs and Ada for the Virtual Design Machine (VDM) [17]. In addition, we are examining the requirements and implications of DIF-based support for other tools that have the ability to map dataflow models to efficient hardware / software implementations.

The DIF package acts as an intermediate layer between abstract dataflow models and different practical implementations. It takes the responsibility of realizing dataflow graphs and performing dataflow-based algorithms. DIF exporting and importing tools automate the process of exporting DSP applications from design tools to DIF specifications and importing them back to design tools. Automating the exporting and importing processes between DIF and design tools provides the DSP design industry a useful front-end to use DIF and the DIF package. In the next section, we will describe issues involved in such automation, and our approaches to addressing these issues.

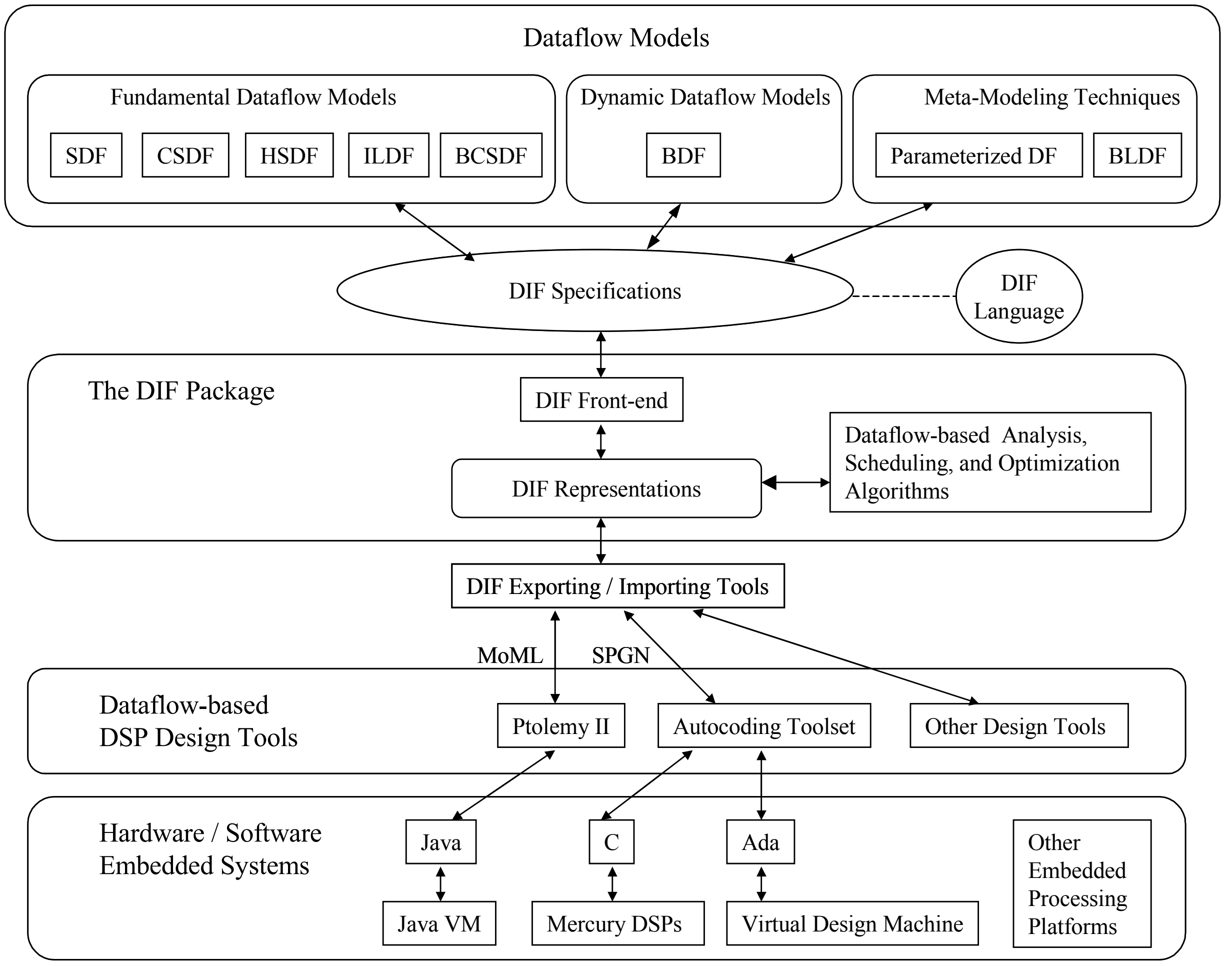


Figure 12. The role of DIF in DSP system design.

6 Exporting and Importing DIF

The DIF language is capable of specifying dataflow semantics of DSP applications in 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 a to design tool's internal representation format. Figure 13 illustrates the exporting and importing mechanisms between DIF and design tools.

When exporting, parsing design tools' specification formats and then directly formulating the corresponding DIF specifications is usually not an efficient way. In contrast, DIF provides a complete set of classes for representing dataflow graphs in a well-designed, object-oriented realization. Hence, instead of parsing and directly formulating equivalent DIF language code, mapping design tools' graphical representations 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 of all, graph topologies and hierarchical structures of DSP applications must be captured in order to completely represent their dataflow semantics. Furthermore, actors' computations, parameters, and 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 DIF-Ptolemy exporting and importing capabilities that we have developed.

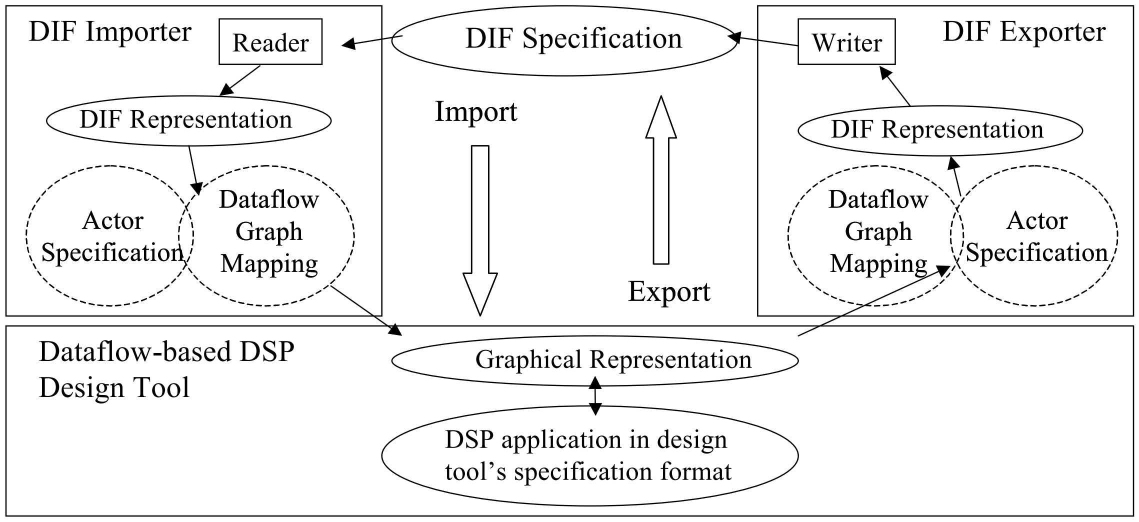


Figure 13. Exporting and Importing Mechanism.

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.

Let's take exporting Ptolemy II to DIF as an example to explain problems in dataflow graph mapping. Ptolemy II has the *AtomicActor* class for representing DSP computations (associated with primitive dataflow nodes) and the *CompositeActor* class for representing subgraphs. It uses the *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, problems arise when mapping Ptolemy II graphical representations to DIF representations. First, based on the formal definition of nodes in dataflow models, they do not have ports to distinguish interfaces. Second, edges in formal dataflow graphs cannot support multiple destinations in contrast to Ptolemy II *Relation*s. Third, the *multiport* property in Ptolemy II does not match 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 still 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 comprehensively traverse graphical representations in a design tool and then map the modeling components encountered to 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 of the algorithm developed for mapping Ptolemy's graphical representations to DIF representations. First, *AtomicActor*s are represented by nodes and *CompositeActor*s are represented by hierarchies. Single-source-single-destination *Relation*s are represented by edges. For a multiple-destination *Relation*, a fork actor (which is described in Section 6.3) and several edges are used to represent it without losing any dataflow properties. A Ptolemy II actor's *IOPort*s and the corresponding connections are specified as actor attributes. Even for a *multiport IOPort*, multiple connections can still be listed as an actor attribute.

6.2 Specifying Actors

In dataflow analysis [15], a node may be viewed as a functional unit associated with a weight that consumes/produces certain numbers of tokens when executing. Usually, dataflow-based analysis and scheduling techniques are based on production rates, consumption rates, edge delays, and various node weight information and other edge weight information (e.g., node execution times or execution time distributions, and the inter-processor communication cost associated with an edge if its source and sink are mapped to different processors in a multiprocessor target). Thus, the detailed computation performed by a node is irrelevant to many dataflow-based analyses. However, the computation (such as an FFT operation) and attributes (such as the order of the FFT) associated with a node is essential during implementation. To avoid confusion between the viewpoints of nodes in dataflow analyses versus in hardware/software implementations, we henceforth use the term *node* for the former context, and we use the term *actor* to refer to a node with specified computation and other implementation-related attributes (for the latter context).

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 be preserved. The *actor block* is newly added to DIF language version 0.2 for actor specification. The DIF language syntax for the actor block is described in Section 3.10. Note that for most dataflow-based analysis and scheduling techniques, the DIF language syntax without the actor block is sufficient.

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 their 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 attributes PARAMETER, INPUT, and OUTPUT in DIF indicate the parameters and interface connections of an actor. A full DIF actor block for the Ptolemy FFT actor is presented in Figure 14. 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 incomingEdgeID and outgoingEdgeID connecting to the corresponding IOPorts.

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

Figure 14. The DIF actor specification for the Ptolemy FFT actor.

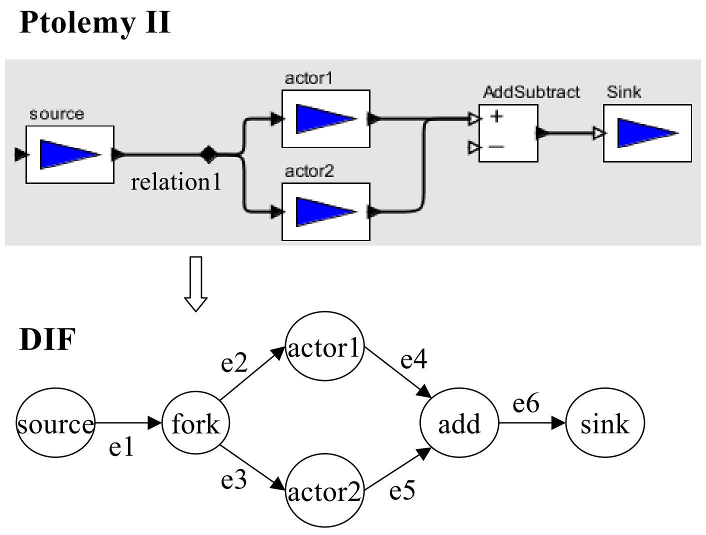
```
actor nodeID {
   computation = "D_FFT";
   N = integerValue or integerParameterID;
   X = incomingEdgeID;
   Y = outgoingEdgeID;
}
```

Figure 15. The DIF actor specification for the D_FFT domain primitive.

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 of information in leaving them out. The corresponding DIF specification for the D_FFT domain primitive is presented in Figure 15.

6.3 The Fork Actor

The fork actor is introduced in DIF as a special built-in actor. It can have one and only 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 is widely used in dataflow. For example, if a stream of data tokens is required to be "broadcast" to multiple destinations, the fork actor can be used for this purpose. The built-in DIF computation associated with the fork actor is called *dif.fork*.



```
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 16. Mapping the Ptolemy II graphical representation to a DIF representation and the corresponding DIF specification.

In dataflow theory, an edge is a data path from a source node to a sink node. It cannot be associated with multiple sink nodes. But the *Relation* in Ptolemy II can have multiple destinations. In order to export Ptolemy's graphical representations to DIF representations, the graph mapping algorithm must be able to 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 sink nodes, we can represent the Ptolemy *Relation* and preserve the same dataflow semantics while using the formal dataflow representations in DIF.

Figure 16 illustrates how the fork actor and actor specification solve the Ptolemy *Relation* and Ptolemy *multiport* problems.

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 the most important feature. Figure 13 illustrates our proposed exporting and importing mechanism. First, a dataflow graph mapping algorithm must be properly designed for the specific design tool that is being used. Then a DIF exporter is implemented for that design tool based on the graph mapping algorithm. It must be able to convert the graphical representation format in that tool to a corresponding DIF representation. Actor specification is also required to preserve the full functionality of actors. 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, the DIF importer can read the DIF specification and generate the DIF representation. Then, based on a "reverse graph mapping algorithm" and actor specification, the DIF importer is able to construct the graphical representation in the design tool while preserving the same functionality of the original DSP application.

The DIF exporter and DIF importer for Ptolemy II are implemented according to the exporting and importing mechanism described above. With these software components, a DSP application in Ptolemy II can be exported to a DIF specification and then be imported back to a Ptolemy MoML specification with all functionality preserved. Such an equivalent result from round-trip translation validates the correctness of the implemented strategies and general methods in DIF for dataflow graph mapping and actor specification.

7 Porting DSP Applications

DIF is proposed to be a standard language for specifying dataflow graphs in all well-defined dataflow models. One of the original goals was to transfer information associated with DSP applications across different dataflow-based design tools. This goal was demonstrated in the first version of DIF [9, 13].

In the development of DIF version 0.2, we have further explored this direction and developed a new and significantly improved approach for porting dataflow-based DSP applications across 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 like SAR across them is a useful demonstration of the DIF porting mechanism.

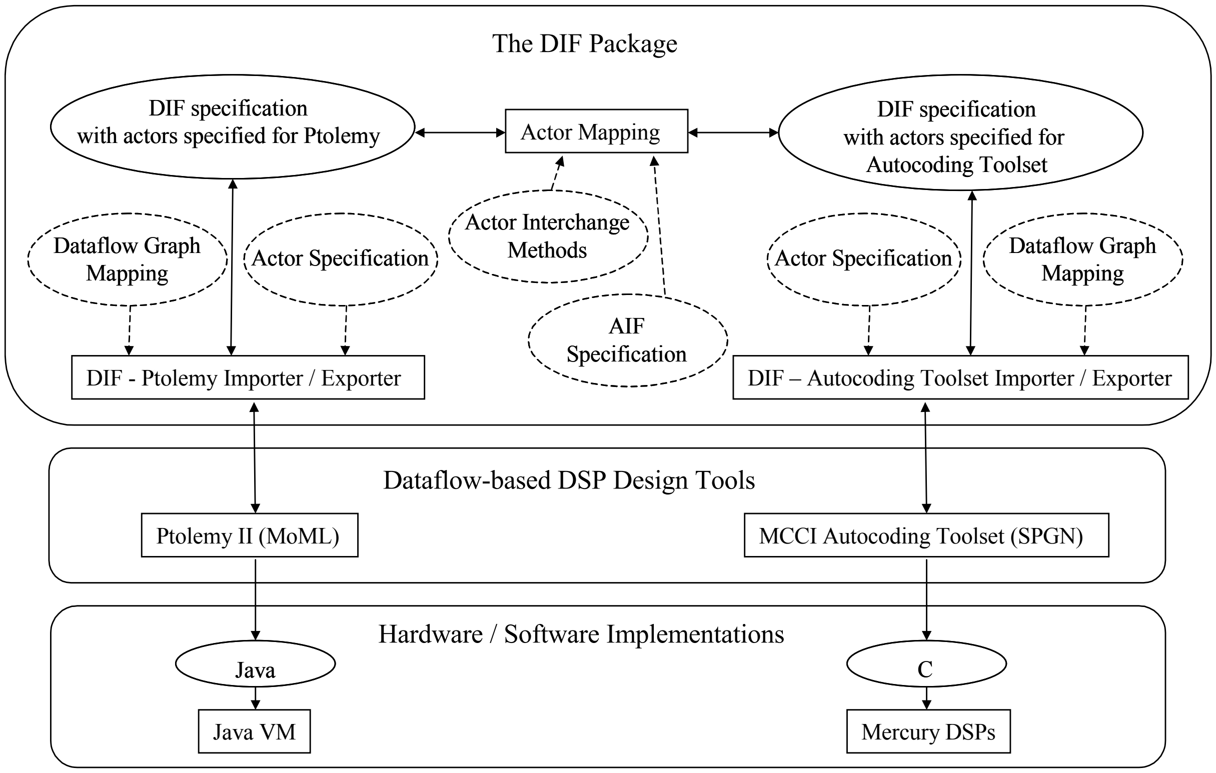

The DIF Package
DIF specification with actors specified for Ptolemy
Actor Mapping
DIF specification with actors specified for Autocoding Toolset
Dataflow Graph Mapping
Actor Specification
Actor Interchange Methods
Actor Specification
Dataflow Graph Mapping
AIF Specification
DIF - Ptolemy Importer / Exporter
DIF – Autocoding Toolset Importer / Exporter
Dataflow-based DSP Design Tools
Ptolemy II (MoML)
MCCI Autocoding Toolset (SPGN)
Hardware / Software Implementations
Java
C
Java VM
Mercury DSPs

Figure 17. The DIF Porting Mechanism.

7.1 The DIF Porting Mechanism

Figure 17 illustrates our proposed porting mechanism. It consists of three major steps: exporting, actor mapping, and importing. Let us take porting from the Autocoding Toolset to Ptolemy II as an example and introduce the porting mechanism in detail.

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, 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 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, Ptolemy does not provide a matrix transpose computation but the Autocoding Toolset does. 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 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 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 construct a subgraph such that the subgraph functionality is compatible with the source actor. 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 17 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. 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. The following subsections 7.3.1 and 7.3.2 introduce the AIF syntax and the SableCC grammar for the Actor Interchange Format is presented in Appendix B.

7.3.1 The Actor-to-Actor Mapping Block

```
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;
}
```

In the first line, the keyword *actor* indicates the actor-to-actor mapping. 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). 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 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 introduced in section 3.11. 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 3.8 and Section 3.10.

7.3.2 Actor to Subgraph Mapping Block

```
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;
   }
}
```

The keyword *graph* in this context indicates the actor-to-subgraph mapping. 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 specify 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 the Java reflection package.

There are three built-in actor interchange methods in the DIF package: 1. *ifExpression("expression")*: this method evaluates the boolean expression and returns true or false; 2. *assign("expression")*: this method evaluates the input expression and returns the evaluated value; 3. *conditionalAssign("valueExpression", "conditionalExpression")*: this method returns the value of valueExpression if the 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 has parameter *X* for data input, parameter *Y* for data output, parameter *N* for FFT length, and parameter *FI* for indicating an FFT or IFFT operation. On the other hand, the Ptolemy 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, parameter the *N* can be mapped to the parameter *order* only when *N = 2^order* is satisfied, where *N* and *order* are integers. The actor interchange specification for mapping the FFT operation from the Autocoding Toolset to Ptolemy II is presented in Figure 18.

```
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 18. The actor interchange specification of mapping the FFT operation.

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 *FI == 0* and returns true. The parameter *order* of the Ptolemy FFT actor is assigned to *log(N) / log(2)* if *log(N) / log(2)* is an integer. Therefore, the actor interchange method *conditionalAssign* evaluates and returns *log(N) / log(2)* if *(log(N)/log(2)) - rint(log(n)/log(2)) == 0* is true, where *rint()* is a round to nearest integer function. Note that if *(log(N)/log(2)) - rint(log(n)/log(2)) == 0* is false, *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*.

```
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 19. The actor interchange specification of actor-to-subgraph mapping of IFFT operation.

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 FFT actor does not support this function. Furthermore, there is a factor of *N* difference between the Autocoding Toolset FFT domain primitive performing the IFFT operation and the Ptolemy IFFT actor. One way to solve this problem is to create a new FFT actor in Ptolemy, but it is rather time-consuming. 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 to the source actor.

The actor interchange specification in Figure 19 illustrates how to map a D_FFT domain primitive with the IFFT operation and selective output length to a Ptolemy subgraph. If a D_FFT domain primitive outputs only part of its sequence, i.e., parameter *N* is not equal to parameter *M*, other Ptolemy actors are involved to extract part of the output sequence of the FFT or IFFT actors. As a result, when *FI == 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*.

7.6 Conclusion

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 17.

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 certain actor mismatch problems because users can collect several 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 an industrial standard library for providing an actor functional interface can further facilitate the porting process. Even with a standard library, the Actor Interchange Format is still essential in mapping actors between tools and the standard library. This is a useful direction for further study in the DIF project.

8 SAR Example

In this section, we demonstrate porting a synthetic aperture radar (SAR) benchmark application from the Autocoding Toolset to Ptolemy II. This demonstration shows the effectiveness the porting mechanisms developed through DIF and AIF. The synthetic aperture radar system examined in this section was used as a benchmark in the Rapid Prototyping of Application Specific Signal Processors (RASSP) program sponsored by DARPA [17]. It represents one type of application where the processing is rather simple but the data rate is extremely high.

8.1 The SAR Application in the MCCI Autocoding Toolset

Figure 20 shows the SAR Functional Requirement developed in the MCCI Autocoding Toolset. Figure 20.(a) illustrates the top-level coarse-grain dataflow graph, *SAR_FR*. The SAR system consists of two major building blocks: range processing and azimuth processing. Passed in through the *SAR_IN* input queue, data samples are processed by node *RANGE* and node *AZIMUTH*, then they are sent to the *SAR_OUT* output queue. Node *RANGE* and node *AZIMUTH* in the *SAR_FR* graph represent the *RNG_FR* subgraph in Figure 20.(b) and the *AZI_FR* subgraph in Figure 20.(c), respectively.

Figure 20.(b) illustrates range processing in the *RNG_FR* subgraph. It consists of four nodes. Node *PAD* pads 16 zero-valued samples to the end of each 2032-sample row. Node *WEIGHT* multiplies each padded range row by a *TAYLOR_WTS* weighting sequence containing 2048 weighting values. Node *COMPRESS* performs a 2048 point Fast Fourier Transform on each range row. Node *COMPENSATE* multiplies the transformed data by the radar cross-section compensation *RCS_WTS* sequence containing 2048 compensating values.

Figure 20.(c) illustrates the azimuth processing in the *AZI_FR* subgraph. It consists of four nodes as follows. Node *CORNERTURN* corner-turns a 1024-by-2048 matrix by using matrix transpose. Node *FFT* performs a 1024-point Fast Fourier Transform on each row of the transposed matrix. Node *CONVL* multiplies each transformed row by a convolution kernel *AZ_KERN* sequence containing 1024 data values. Node *IFFT* performs an Inverse Fast Fourier Transform on the convolution result and outputs only the last 512 samples.

8.2 Porting the SAR Application to Ptolemy II

Appendix C presents the DIF specification of the Autocoding Toolset SAR application showed in Figure 20. With the actor interchange specification presented in Appendix D and the actor interchange methods developed in the DIF package, our actor mapping mechanism can translate the

DIF specification in Appendix C to the DIF specification for Ptolemy actors, which is presented in Appendix E. Finally, the DIF-Ptolemy exporter/importer imports the DIF specification in Appendix E to Ptolemy II. The ported graphical representation in Ptolemy II is showed in Figure 21.

Figure 21.(a) represents the top-level coarse grain graph of the SAR application in Ptolemy II. The supernodes (blocks with red borders) RNG_FR and AZI_FR represent the range processing subgraph and the azimuth processing subgraph, respectively. Figure 21.(b) is the range processing RNG_FR graph and Figure 21.(c) is the azimuth processing graph. Node *IFFT* in Figure 20.(c) outputs only half of the IFFT sequence and there is a factor of *N* difference; actor-to-subgraph mapping is used to solve these actor mismatch problems. The node *IFFT* in Figure 20.(c) is mapped to the IFFT_SUBGRAPH in Figure 21.(d).

The MCCI Autocoding Toolset has I/O procedures specified outside of its graph specifications. As a result, we manually added I/O actors to feed data samples as well as coefficients into the SAR graph, and to write and display the results. Figure 22 shows the SAR application in Ptolemy II after adding I/O actors. The supernode SAR_FR in Figure 22 represents the top-level SAR in Figure 21.(a). Other actors in Figure 22 are used to read input samples, read coefficients, write to a file, and display the absolute value of the output waveform.

The ported SAR benchmark application in Ptolemy II works correctly. Figure 23 shows the output waveform in Ptolemy II. Figure 24 compares the output samples generated by Ptolemy II with those generated by the Autocoding Toolset, and reveals that the simulation results are the same except for tolerable precision errors.

9 Conclusions and Future Work

In this report, we have introduced the DIF language version 0.2, the DIF package, and the supported dataflow models. We described our approach to automate the exporting and importing processes. Finally, we developed the DIF porting mechanism and demonstrated it through a detailed example of porting the SAR benchmark application between the MCCI Autocoding Toolset and Ptolemy II.

In ongoing and future work on the DIF project, we are extending the DIF language and the DIF package to accommodate advanced dataflow semantics such as various additional forms of dynamic graph elements and multi-mode graphs. Another useful direction for further work is integration with industrial dataflow-based design tools by combining algorithms in the DIF package with their software synthesis and code generation techniques.

10 Acknowledgements

This research was sponsored in part by DARPA (contract #F30602-01-C-0171, through the USC Information Sciences Institute and contract #DAAH01-03-C-R236 via the U.S. Army Aviation and Missile Command and through MCCI), and the Semiconductor Research Corporation (contract #2001-HJ-905).

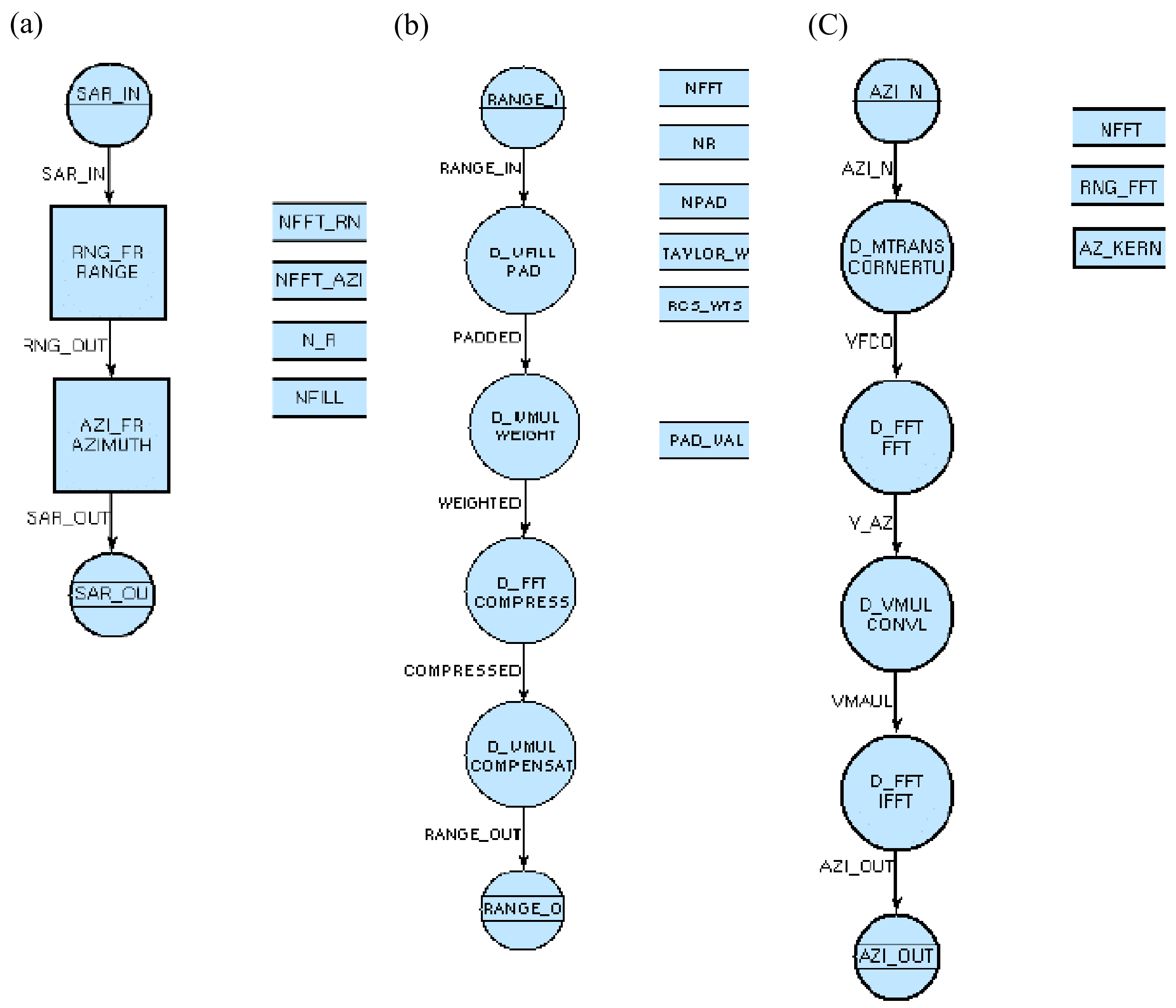
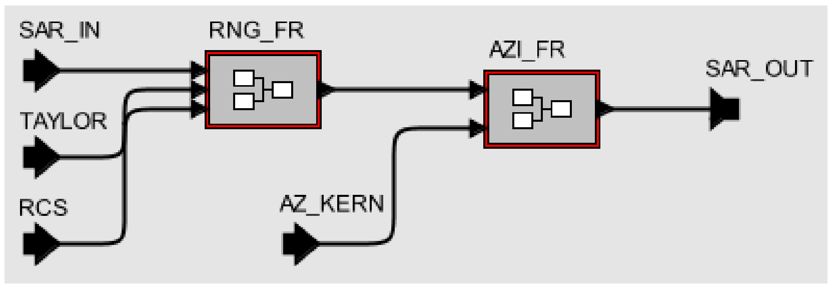
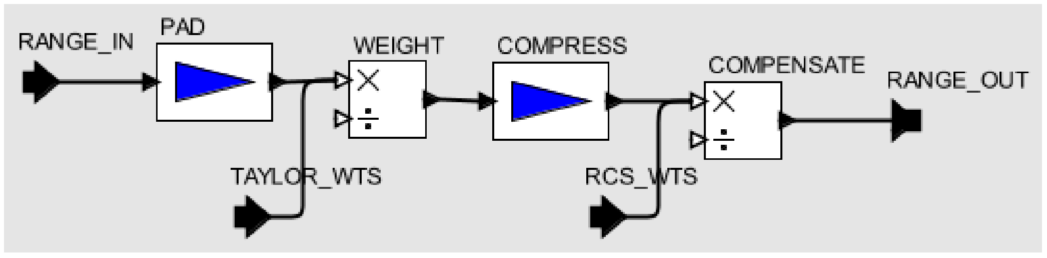
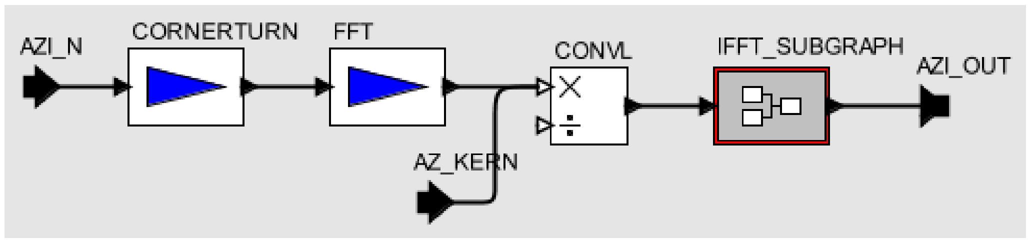


Figure 20. The SAR benchmark application in the MCCI Autocoding Toolset.
(a) SAR_FR graph. (b) RNG_FR graph. (c) AZI_FR graph.

(a)


(b)


(c)


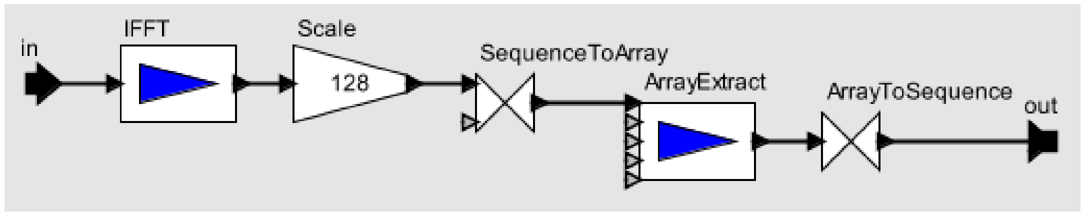
(d)


Figure 21. The SAR benchmark application in Ptolemy II.
(a) SAR_FR graph. (b) RNG_FR graph. (c) AZI_FR graph. (d) IFFT_SUBGRAPH graph

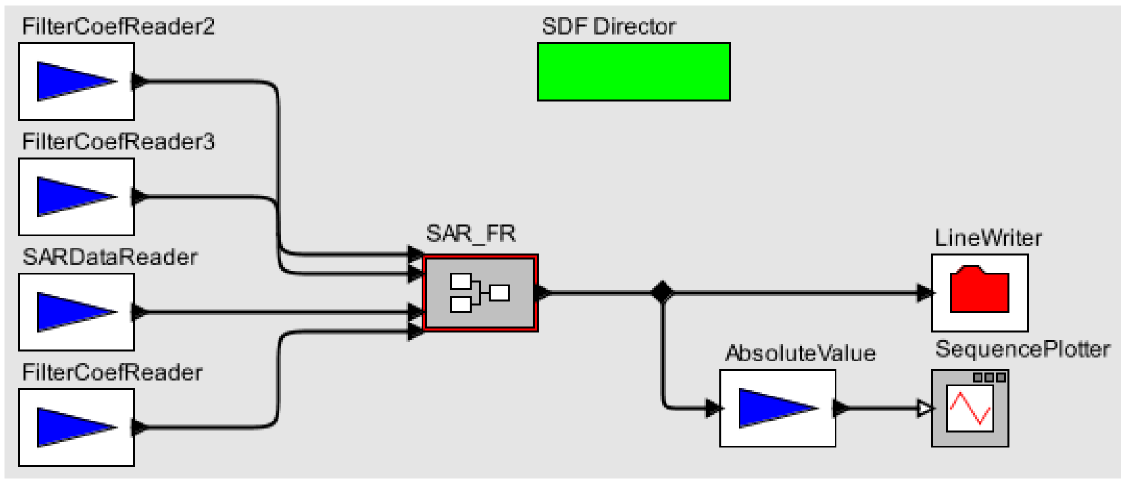


Figure 22. The SAR benchmark application in Ptolemy II after adding I/O actors

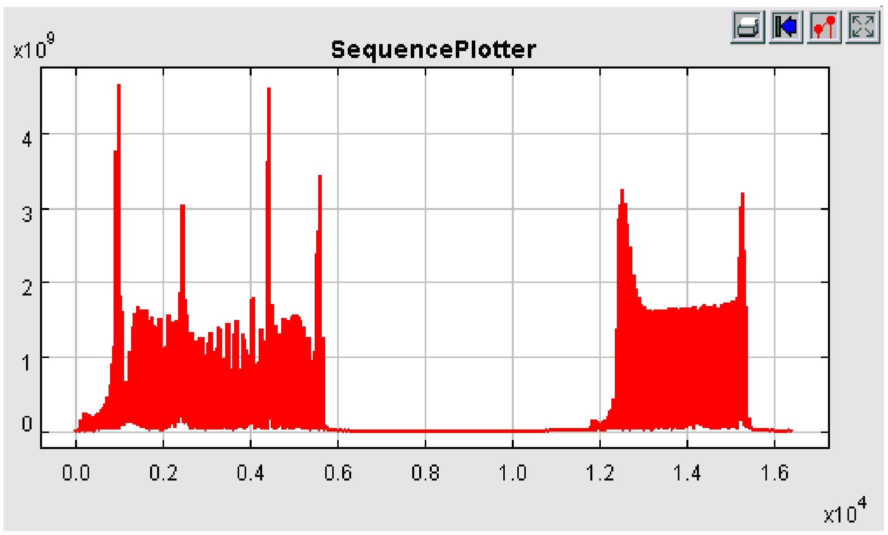
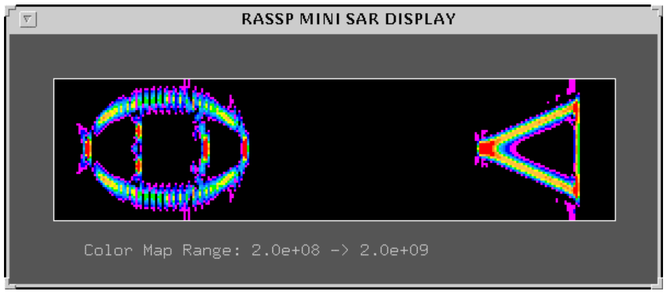


Figure 23. Simulation result of the SAR benchmark application in Ptolemy II.

Ptolemy(PGM Export)

```
1.113328370318E9, -5.672582199684E8
1.686243152456E9, -1.132239286739E9
2.280892492213E9, -1.837179778052E9
2.787030647091E9, -2.565079199379E9
3.121469726315E9, -3.124321013999E9
3.235633491442E9, -3.339997173742E9
3.126105298721E9, -3.132702116709E9
2.795907223687E9, -2.578937710771E9
2.292518065694E9, -1.852489499236E9
1.698661416987E9, -1.145532955647E9
```



MCCI(DIF Import)

```
1.11334E+09, -5.67194E+08
1.68657E+09, -1.13206E+09
2.28101E+09, -1.83712E+09
2.78720E+09, -2.56485E+09
3.12169E+09, -3.12429E+09
3.23570E+09, -3.33972E+09
3.12633E+09, -3.13268E+09
2.79604E+09, -2.57867E+09
2.29266E+09, -1.85242E+09
1.69888E+09, -1.14531E+09
```

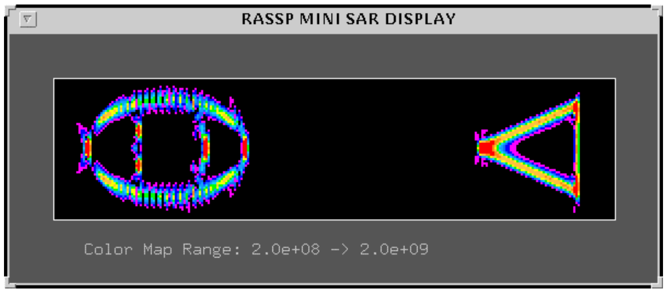


Figure 24. Simulation results of the SAR benchmark application in Ptolemy II and the MCCI Autocoding Toolset.

References

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

[2] S. S. Bhattacharyya. 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.

[3] 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.

[4] 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.

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

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *Proc. ICASSP*, pages 3255-3258, May 1995.

[7] 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.

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

[9] 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.

[10] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, H. Zheng (eds.). *Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)*. Technical Memorandum UCB/ERL M03/27, University of California, Berkeley, CA USA 94720, July 16, 2003.

[11] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, H. Zheng, (eds.). *Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture).* Technical Memorandum UCB/ERL M03/28, University of California, Berkeley, CA USA 94720, July 16, 2003.

[12] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, H. Zheng (eds.). *Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)*. Technical Memorandum UCB/ERL M03/29, University of California, Berkeley, CA USA 94720, July 16, 2003.

[13] 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.

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

[15] K. K. Parhi. *VLSI Digital Signal Processing Systems, Design and Implementation*, John Wiley, 1999.

[16] C. B. Robbins. *Using The MCCI Autocoding Toolset Overview*, Document Version 0.98a, Management, Communications & Control, Inc.

[17] C. B. Robbins. *Using The MCCI Autocoding Toolset Tutorial*. Document Version 0.9a, Management, Communications & Control, Inc.

[18] 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.

Appendix A

The SableCC (version 2.16.2) grammar of the Dataflow Interchange Format.

```
Package mapss.dif.language.sablecc;

Helpers
  all = [0 .. 127];
  digit = ['0' .. '9'];
  non_digit = [[['a' .. 'z'] + ['A' .. 'Z']] + '_'];
  double = ( '+' | '-' )? (digit*) '.' (digit+)
           ( ('e' | 'E') ( '+' | '-' )? digit+ )?;
  integer = ( '-' )? digit+;

  tab = 9;
  cr = 13;
  lf = 10;
  eol = cr lf | cr | lf; // This takes care of different platforms

  not_cr_lf = [all -[cr + lf]];
  not_star = [all -'*'];
  not_star_slash = [not_star -'/'];

  short_comment = '//' not_cr_lf* eol;
  long_comment = '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';
  comment = long_comment | short_comment;

  simple_escape_sequence = '\' ''' | '\"' | '\\' |
    '\b' | '\f' | '\n' | '\r' | '\t';
  octal_digit = ['0' .. '7'];
  octal_escape_sequence = '\' octal_digit octal_digit? octal_digit?;
  hexadecimal_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];
  hexadecimal_escape_sequence = '\x' hexadecimal_digit+;
  escape_sequence = simple_escape_sequence | octal_escape_sequence |
  hexadecimal_escape_sequence;
  s_char = [all -['"' + ['$' + ['\' + [10 + 13]]]]] | escape_sequence;
  s_char_sequence = s_char*;
  string = '"' s_char_sequence '"';
  string_identifier = '$' s_char_sequence '$';

Tokens

  blank = (' ' | tab | eol);
  comment = comment;

  l_bkt = '{';
  r_bkt = '}';
  l_par = '(';
  r_par = ')';
  l_sqr = '[';
  r_sqr = ']';
  semicolon = ';';
  colon = ':';
  comma = ',';
  s_qte = ''';
  plus = '+';
  equal = '=';
  dot = '.';

  graph = 'graph';
  attribute = 'attribute';
  basedon = 'basedon';
  interface = 'interface';
  parameter = 'parameter';
  refinement = 'refinement';
  topology = 'topology';
  actor = 'actor';
  inputs = 'inputs';
  outputs = 'outputs';
  nodes = 'nodes';
  edges = 'edges';

  integer = integer;
  double = double;
  true = 'true';
  false = 'false';
  string = string;
  string_tail = '+' (' ' | eol | tab)* string;

  identifier = non_digit (digit | non_digit)*;
   dot_identifier  =  non_digit  (digit  |  non_digit)*  ('.'  non_digit  (digit  |
non_digit)* )+;
  string_identifier = string_identifier;

Ignored Tokens

  blank,
  comment;

Productions

  graph_list = graph_block*;
  graph_block = identifier name l_bkt block* r_bkt;
  block =
    {basedon}                   basedon basedon_body |
    {topology}                  topology topology_body |
    {interface}                 interface interface_body |
    {parameter}                 parameter parameter_body |
    {refinement}                refinement refinement_body |
    {builtin_attribute}         identifier attribute_body |
    {user_defined_attribute}    attribute name attribute_body |
    {actor}                     actor name actor_body;

  name = {identifier} identifier | {string_identifier} string_identifier;

  /*************************************
   * Definitions for basedon block:
```

```
 */

basedon_body = l_bkt basedon_expression r_bkt;
basedon_expression = name semicolon;

/************************************
 * Definitions for topology block:
 */

topology_body = l_bkt topology_list* r_bkt;
topology_list =
  {nodes}  nodes equal name node_identifier_tail* semicolon |
  {edges}  edges equal edge_definition edge_definition_tail* semicolon ;

node_identifier_tail = comma name;
edge_definition = [edge]:name l_par
                  [source]:name comma
                  [sink]:name r_par;
edge_definition_tail = comma edge_definition;

/************************************
 * Definitions for interface block:
 */

interface_body = l_bkt interface_expression* r_bkt;
interface_expression =
  {input}   inputs equal port_definition port_definition_tail* semicolon |
  {output}  outputs equal port_definition port_definition_tail* semicolon;

port_definition = {plain} name |
                  {node}  [port]:name colon [node]:name;
port_definition_tail = comma port_definition;

/************************************
 * Definitions for parameter block:
 */

parameter_body = l_bkt parameter_expression* r_bkt;
parameter_expression =
  {value}     name equal value semicolon |
  {range}     name colon range_block semicolon |
  {blank}     name semicolon;

range_block = range range_tail*;
range =
  {closed_closed}     l_sqr [left]:number comma [right]:number r_sqr |
  {open_closed}       l_par [left]:number comma [right]:number r_sqr |
  {closed_open}       l_sqr [left]:number comma [right]:number r_par |
  {open_open}         l_par [left]:number comma [right]:number r_par |
  {discrete}          l_bkt number discrete_range_number_tail* r_bkt;
discrete_range_number_tail = comma number;
range_tail = plus range;
number = {double} double | {integer} integer;
```

```
/************************************
 * Definitions for refinement block:
 */

refinement_body = l_bkt refinement_definition refinement_expression* r_bkt;
refinement_definition = [graph]:name equal [node]:name semicolon;
refinement_expression =
  {ports}     [port]:name colon [element]:name semicolon |
  {params}    [subparam]:name equal [param]:name semicolon;

/************************************
 * Definitions for attribute block:
 */

attribute_body = l_bkt attribute_expression* r_bkt;
attribute_expression =
  {value} name? equal value semicolon |
  {reference} [element]:name? equal [reference]:name semicolon |
  {subelement_assign} [trggraph]:name [fst]:dot [trgele]:name equal
                      [srcgraph]:name [snd]:dot [srcele]:name semicolon |
  {idlist} name? equal id_list semicolon;

id_list = name ref_id_tail+;
ref_id_tail = comma name;

/************************************
 * Definitions for actor block:
 */

actor_body = l_bkt actor_expression* r_bkt;
actor_expression =
  {value} name type? equal value semicolon |
  {reference} [argument]:name type? equal [reference]:name semicolon |
  {reflist} name type? equal id_list semicolon;

type =
  {identifier} colon identifier |
  {dot_identifier} colon dot_identifier;

/************************************
 * Definitions for value:
 */

value =
  {integer} integer |
  {double} double |
  {complex} l_par [real]:double comma [imag]:double r_par |
  {int_matrix} l_sqr int_row int_row_tail* r_sqr |
  {double_matrix} l_sqr double_row double_row_tail* r_sqr |
  {complex_matrix} l_sqr complex_row complex_row_tail* r_sqr |
  {string} concatenated_string_value |
  {boolean} boolean_value |
  {array} l_bkt value value_tail* r_bkt;
```

```
int_row = integer integer_tail*;
integer_tail = comma integer;
int_row_tail = semicolon int_row;

double_row = double double_tail*;
double_tail = comma double;
double_row_tail = semicolon double_row;

complex = l_par [real]:double comma [imag]:double r_par;
complex_row = complex complex_tail*;
complex_tail = comma complex;
complex_row_tail = semicolon complex_row;

concatenated_string_value = string string_tail*;

boolean_value =
  {true} true |
  {false} false;

value_tail = comma value;
```

Appendix B

The SableCC (version 2.16.2) grammar of the Actor Interchange Format.

```
Package mapss.dif.aif.sablecc;

Helpers
  all = [0 .. 127];
  digit = ['0' .. '9'];
  non_digit = [[['a' .. 'z'] + ['A' .. 'Z']] + '_'];
  double = ( '+' | '-' )? (digit*) '.' (digit+)
           ( ('e' | 'E') ( '+' | '-' )? digit+ )?;
  integer = ( '-' )? digit+;

  tab = 9;
  cr = 13;
  lf = 10;
  eol = cr lf | cr | lf; // This takes care of different platforms

  not_cr_lf = [all -[cr + lf]];
  not_star = [all -'*'];
  not_star_slash = [not_star -'/'];

  short_comment = '//' not_cr_lf* eol;
  long_comment = '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';
  comment = long_comment | short_comment;

  simple_escape_sequence = '\' ''' | '\"' | '\\' |
    '\b' | '\f' | '\n' | '\r' | '\t';
  octal_digit = ['0' .. '7'];
  octal_escape_sequence = '\' octal_digit octal_digit? octal_digit?;
  hexadecimal_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];
  hexadecimal_escape_sequence = '\x' hexadecimal_digit+;
  escape_sequence = simple_escape_sequence | octal_escape_sequence |
  hexadecimal_escape_sequence;
  s_char = [all -['"' + ['$' + ['\' + [10 + 13]]]]] | escape_sequence;
  s_char_sequence = s_char*;
  string = '"' s_char_sequence '"';
  string_identifier = '$' s_char_sequence '$';

Tokens

  blank = (' ' | tab | eol);
  comment = comment;

  l_bkt = '{';
  r_bkt = '}';
  l_par = '(';
  r_par = ')';
  l_sqr = '[';
  r_sqr = ']';
  semicolon = ';';
  colon = ':';
  comma = ',';
  s_qte = ''';
  plus = '+';
  equal = '=';
  dot = '.';
  map_to = '->';
  map_from = '<-';
  given_that = '|';

  graph = 'graph';
  interface = 'interface';
  topology = 'topology';
  actor = 'actor';
  inputs = 'inputs';
  outputs = 'outputs';
  nodes = 'nodes';
  edges = 'edges';

  integer = integer;
  double = double;
  true = 'true';
  false = 'false';
  string = string;
  string_tail = '+' (' ' | eol | tab)* string;

  identifier = non_digit (digit | non_digit)*;
  dot_identifier = non_digit (digit | non_digit)*
                   ('.' non_digit (digit | non_digit)* )+;
  string_identifier = string_identifier;

Ignored Tokens

  blank,
  comment;

Productions

  aif_list = aif_block*;
  aif_block = {actor} actor [trg]:type map_from [src]:type
                      method_expression? l_bkt attribute_body* r_bkt |
              {graph} graph [trg]:type map_from [src]:type
                      method_expression? l_bkt block* r_bkt;

  type = {identifier} identifier |
         {dot_identifier} dot_identifier;
  method_expression = given_that identifier
                      l_par argument argument_tail* r_par;
  argument = {id} identifier |
             {value} value;
  argument_tail = comma argument;

  /*************************************
   * Definitions for attribute body:
   */
```

```
attribute_body = {mapping} attribute_mapping |
                 {assign} attribute_assign;
attribute_assign = attribute equal value semicolon;
attribute_mapping = {single} [trg]:attribute map_from [src]:attribute
                              method_expression? semicolon |
                    {multi_to_one} attribute map_from attributes semicolon |
                    {one_to_multi} attributes map_from attribute semicolon;

attributes = attribute attribute_tail+;
type_expression = colon type;
attributes = attribute attribute_tail+;
attribute_tail = comma attribute;

/************************************
 * Definitions for block:
 */

block = {topology} topology topology_body |
        {interface} interface interface_body |
        {actor} actor name actor_body;

name = {identifier} identifier |
       {string_identifier} string_identifier;

/************************************
 * Definitions for topology block:
 */

topology_body = l_bkt topology_list* r_bkt;
topology_list =
  {nodes}  nodes equal name node_identifier_tail* semicolon |
  {edges}  edges equal edge_definition edge_definition_tail* semicolon ;

node_identifier_tail = comma name;
edge_definition = [edge]:name l_par
                  [source]:name comma
                  [sink]:name r_par;
edge_definition_tail = comma edge_definition;

/************************************
 * Definitions for interface block:
 */

interface_body = l_bkt interface_expression* r_bkt;
interface_expression =
  {input}   inputs equal port_definition port_definition_tail* semicolon |
  {output}  outputs equal port_definition port_definition_tail* semicolon;

port_definition = {plain} name port_mapping? |
                  {node}  [port]:name colon [node]:name port_mapping?;
port_definition_tail = comma port_definition;
port_mapping = map_from attribute;
```

```
  /************************************
   * Definitions for actor block:
   */

  actor_body = l_bkt actor_expression* r_bkt;
  actor_expression =
    {value} name type_expression? equal value semicolon |
    {reference} [argument]:name type_expression? equal
                [reference]:name semicolon |
    {map} name type_expression? map_from attribute
          method_expression? semicolon |
    {multi_map} name type_expression? map_from attributes semicolon |
    {reflist} name type_expression? equal id_list semicolon;

  id_list = name ref_id_tail+;
  ref_id_tail = comma name;

/************************************
   * Definitions for value:
   */

  value =
    {integer} integer |
    {double} double |
    {complex} l_par [real]:double comma [imag]:double r_par |
    {int_matrix} l_sqr int_row int_row_tail* r_sqr |
    {double_matrix} l_sqr double_row double_row_tail* r_sqr |
    {complex_matrix} l_sqr complex_row complex_row_tail* r_sqr |
    {string} concatenated_string_value |
    {boolean} boolean_value |
    {array} l_bkt value value_tail* r_bkt;

  int_row = integer integer_tail*;
  integer_tail = comma integer;
  int_row_tail = semicolon int_row;

  double_row = double double_tail*;
  double_tail = comma double;
  double_row_tail = semicolon double_row;

  complex = l_par [real]:double comma [imag]:double r_par;
  complex_row = complex complex_tail*;
  complex_tail = comma complex;
  complex_row_tail = semicolon complex_row;

  concatenated_string_value = string string_tail*;

  boolean_value =
    {true} true |
    {false} false;

  value_tail = comma value;
```

Appendix C

The DIF specification of the MCCI SAR benchmark application.

```
dif RNG_FR {
  topology {
    nodes = PAD, WEIGHT, COMPRESS, COMPENSATE;
    edges = PADDED (PAD, WEIGHT),
            WEIGHTED (WEIGHT, COMPRESS),
            COMPRESSED (COMPRESS, COMPENSATE);
  }
  interface {
    inputs = RANGE_IN : PAD, TAYLOR_WTS : WEIGHT, RCS_WTS : COMPENSATE;
    outputs = RANGE_OUT : COMPENSATE;
  }
  parameter {
    NFFT;
    NR;
    NPAD;
    PAD_VAL = (0.0, 0.0);
  }
  actor PAD {
    computation = "D_VFILL" ;
    N = NR;
    P = NPAD;
    V = PAD_VAL;
    X = RANGE_IN;
    Y = PADDED;
  }
  actor WEIGHT {
    computation = "D_VMUL";
    N = NFFT;
    X = PADDED;
    Y = TAYLOR_WTS;
    Z = WEIGHTED;
  }
  actor COMPRESS {
    computation = "D_FFT";
    N = NFFT;
    FI = 0;
    X = WEIGHTED;
    Y = COMPRESSED;
  }
  actor COMPENSATE {
    computation = "D_VMUL";
    N = NFFT;
    X = COMPRESSED;
    Y = RCS_WTS;
    Z = RANGE_OUT;
  }
}

dif AZI_FR {
  topology {
    nodes = CORNERTURN, FFT, CONVL, IFFT;
    edges = YFCO (CORNERTURN, FFT),
            Y_AZ (FFT, CONVL),
            VMAUL (CONVL, IFFT);
  }
  interface {
    inputs = AZI_N : CORNERTURN, AZ_KERN : CONVL;
    outputs = AZI_OUT : IFFT;
  }
  parameter {
    NFFT;
    RNG_FFT;
  }
  actor CORNERTURN {
    computation = "D_MTRAN";
    M = NFFT;
    N = RNG_FFT;
    X = AZI_N;
    Y = YFCO;
  }
  actor FFT {
    computation = "D_FFT";
    N = NFFT;
    FI = 0;
    X = YFCO;
    Y = Y_AZ;
  }
  actor CONVL {
    computation = "D_VMUL";
    N = NFFT;
    X = Y_AZ;
    Y = AZ_KERN;
    Z = VMAUL;
  }
  actor IFFT {
    computation = "D_FFT";
    N = NFFT;
    FI = 1;
    X = VMAUL;
    Y = AZI_OUT;
    M = "NFFT/2";
    B = "(NFFT/2)+1";
  }
}

dif FR_SAR {
  topology {
    nodes = RANGE, AZIMUTH;
    edges = RNG_OUT (RANGE, AZIMUTH);
  }
  interface {
    inputs = SAR_IN : RANGE, TAYLOR : RANGE, RCS : RANGE, AZ_KERN : AZIMUTH;
    outputs = SAR_OUT : AZIMUTH;
  }
  parameter {
    NFFT_RNG = 256;
    NFFT_AZI = 128;
    N_R = 235;
    NFILL = "NFFT_RNG-N_R";
  }
  refinement {
    RNG_FR = RANGE;
    RANGE_IN : SAR_IN;
    RANGE_OUT : RNG_OUT;
    TAYLOR_WTS : TAYLOR;
    RCS_WTS : RCS;
    NFFT = NFFT_RNG;
    NR = N_R;
    NPAD = NFILL;
  }
  refinement {
    AZI_FR = AZIMUTH;
    AZI_N : RNG_OUT;
    AZI_OUT : SAR_OUT;
    AZ_KERN : AZ_KERN;
    NFFT = NFFT_AZI;
    RNG_FFT = NFFT_RNG;
  }
  actor RANGE {
    computation = "SUBGRAPH";
  }
  actor AZIMUTH {
    computation = "SUBGRAPH";
  }
}
```

Appendix D

The actor interchange specification for Autocoding Toolset to Ptolemy II actor mapping.

```
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;
    }
}
```

```
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;
}

actor ptolemy.domains.sdf.lib.IFFT <- D_FFT | ifExpression("FI == 1") {
    order : PARAMETER <- N | conditionalAssign(
        "log(N)/log(2)", "(log(N)/log(2))-rint(log(N)/log(2)) == 0");
    input : INPUT <- X;
    output : OUTPUT <- Y;
}


actor mapss.applications.sar.MatrixTranspose <- D_MTRAN {
    rowN : PARAMETER <- M;
    colN : PARAMETER <- N;
    input : INPUT <- X;
    output : OUTPUT <- Y;
}

actor ptolemy.actor.lib.MultiplyDivide <- D_VMUL {
    multiply : INPUT <- X, Y;
    output : OUTPUT <- Z;
}

actor mapss.applications.sar.SequencePad <- D_VFILL {
    inputLength : PARAMETER <- N;
    outputLength : PARAMETER <- P | assign("P+N");
    padValue : PARAMETER <- V;
    input : INPUT <- X;
    output : OUTPUT <- Y;
}

actor ptolemy.actor.TypedCompositeActor <- SUBGRAPH {}
```

Appendix E

The DIF specification of the ported SAR benchmark application in Ptolemy II.

```
dif IFFT_SUBGRAPH {
    topology {
        nodes = IFFT,
                Scale,
                SequenceToArray,
                ArrayExtract,
                ArrayToSequence;
        edges = e1 (IFFT, Scale),
                e2 (Scale, SequenceToArray),
                e3 (SequenceToArray, ArrayExtract),
                e4 (ArrayExtract, ArrayToSequence);
    }
    interface {
        inputs = in:IFFT;
        outputs = out:ArrayToSequence;
    }
    actor IFFT {
        computation = "ptolemy.domains.sdf.lib.IFFT";
        order : PARAMETER = 7.0;
        input : INPUT = in;
        output : OUTPUT = e1;
    }
    actor Scale {
        computation = "ptolemy.actor.lib.Scale";
        input : INPUT = e1;
        output : OUTPUT = e2;
        factor : PARAMETER = 128;
    }
    actor SequenceToArray {
        computation = "ptolemy.domains.sdf.lib.SequenceToArray";
        input : INPUT = e2;
        output : OUTPUT = e3;
        arrayLength : PARAMETER = 128;
    }
    actor ArrayExtract {
        computation = "ptolemy.actor.lib.ArrayExtract";
        input : INPUT = e3;
        output : OUTPUT = e4;
        sourcePosition : PARAMETER = 64;
        extractLength : PARAMETER = 64;
        destinationPosition : PARAMETER = 0;
        outputArrayLength : PARAMETER = 64;
    }
    actor ArrayToSequence {
        computation = "ptolemy.domains.sdf.lib.ArrayToSequence";
        input : INPUT = e4;
        output : OUTPUT = out;
        arrayLength : PARAMETER = 64;
    }
```

```
}

dif RNG_FR {
    topology {
        nodes = PAD,
                WEIGHT,
                COMPRESS,
                COMPENSATE;
        edges = PADDED (PAD, WEIGHT),
                WEIGHTED (WEIGHT, COMPRESS),
                COMPRESSED (COMPRESS, COMPENSATE);
    }
    interface {
        inputs = RANGE_IN:PAD,
                 TAYLOR_WTS:WEIGHT,
                 RCS_WTS:COMPENSATE;
        outputs = RANGE_OUT:COMPENSATE;
    }
    parameter {
        NFFT;
        NR;
        NPAD;
        PAD_VAL = (0.0,0.0);
    }
    actor PAD {
        computation = "mapss.applications.sar.SequencePad";
        inputLength : PARAMETER = NR;
        outputLength : PARAMETER = 256;
        padValue : PARAMETER = PAD_VAL;
        input : INPUT = RANGE_IN;
        output : OUTPUT = PADDED;
    }
    actor WEIGHT {
        computation = "ptolemy.actor.lib.MultiplyDivide";
        multiply : INPUT = PADDED, TAYLOR_WTS;
        output : OUTPUT = WEIGHTED;
    }
    actor COMPRESS {
        computation = "ptolemy.domains.sdf.lib.FFT";
        order : PARAMETER = 8.0;
        input : INPUT = WEIGHTED;
        output : OUTPUT = COMPRESSED;
    }
    actor COMPENSATE {
        computation = "ptolemy.actor.lib.MultiplyDivide";
        multiply : INPUT = COMPRESSED, RCS_WTS;
        output : OUTPUT = RANGE_OUT;
    }
}

dif AZI_FR {
    topology {
        nodes = CORNERTURN,
                FFT,
                CONVL,
                IFFT;
        edges = YFCO (CORNERTURN, FFT),
                Y_AZ (FFT, CONVL),
                VMAUL (CONVL, IFFT);
    }
    interface {
        inputs = AZI_N:CORNERTURN,
                 AZ_KERN:CONVL;
        outputs = AZI_OUT:IFFT;
    }
    parameter {
        NFFT;
        RNG_FFT;
    }
    refinement {
        IFFT_SUBGRAPH = IFFT;
        in : VMAUL;
        out : AZI_OUT;
    }
    actor CORNERTURN {
        computation = "mapss.applications.sar.MatrixTranspose";
        rowN : PARAMETER = NFFT;
        colN : PARAMETER = RNG_FFT;
        input : INPUT = AZI_N;
        output : OUTPUT = YFCO;
    }
    actor FFT {
        computation = "ptolemy.domains.sdf.lib.FFT";
        order : PARAMETER = 7.0;
        input : INPUT = YFCO;
        output : OUTPUT = Y_AZ;
    }
    actor CONVL {
        computation = "ptolemy.actor.lib.MultiplyDivide";
        multiply : INPUT = Y_AZ, AZ_KERN;
        output : OUTPUT = VMAUL;
    }
    actor IFFT {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
}

dif FR_SAR {
    topology {
        nodes = RANGE,
                AZIMUTH;
        edges = RNG_OUT (RANGE, AZIMUTH);
    }
    interface {
        inputs = SAR_IN:RANGE,
                 TAYLOR:RANGE,
                 RCS:RANGE,
                 AZ_KERN:AZIMUTH;
        outputs = SAR_OUT:AZIMUTH;
    }
    parameter {
        NFFT_RNG = 256;
        NFFT_AZI = 128;
        N_R = 235;
        NFILL = "NFFT_RNG-N_R";
    }
    refinement {
        RNG_FR = RANGE;
        RANGE_IN : SAR_IN;
        TAYLOR_WTS : TAYLOR;
        RCS_WTS : RCS;
        RANGE_OUT : RNG_OUT;
        NFFT = NFFT_RNG;
        NR = N_R;
        NPAD = NFILL;
    }
    refinement {
        AZI_FR = AZIMUTH;
        AZI_N : RNG_OUT;
        AZ_KERN : AZ_KERN;
        AZI_OUT : SAR_OUT;
        NFFT = NFFT_AZI;
        RNG_FFT = NFFT_RNG;
    }
    actor RANGE {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
    actor AZIMUTH {
        computation = "ptolemy.actor.TypedCompositeActor";
    }
}
```