Proceedings of the International Workshop on Rapid System Prototyping, pages 40-46, Montreal, Canada, June 2005.

Porting DSP Applications across Design Tools Using the Dataflow Interchange Format

Chia-Jui Hsu and Shuvra S. Bhattacharyya

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

Abstract

Modeling DSP applications through coarse-grain dataflow graphs is popular in the DSP design community, and a growing set of rapid prototyping tools support such dataflow semantics. Since different tools may be suitable for different phases or generations of a design, it is often desirable to migrate a dataflow-based application model from one prototyping tool to another. Two critical problems in transferring dataflow-based designs across different prototyping tools are the lack of a vendor-independent language for DSP-oriented dataflow graphs, and the lack of an efficient porting methodology. In our previous work, the dataflow interchange format (DIF) [5] has been developed as a standard language to specify mixed-grain dataflow models for DSP systems. This paper presents the augmentation of the DIF infrastructure with a systematic porting approach that integrates DIF tightly with the specific exporting and importing mechanisms that interface DIF to specific DSP design tools. In conjunction with this porting mechanism, this paper also introduces a novel language, called the actor interchange format (AIF), for transferring relevant information pertaining to DSP library components across different tools. Through a case study of a synthetic aperture radar application, we demonstrate the high degree of automation offered by our DIF-based porting approach.

1. Introduction

Dataflow semantics are widely used in many design and rapid prototyping tools for DSP systems [1]. These tools generally support different sets of dataflow models, DSP libraries, and target platforms. Developing or migrating designs across multiple tools often becomes desirable because different tools may have complementary features (e.g., simulation vs. synthesis, hardware vs. software support, etc.), and different generations of designs may be best suited to different types of tools. Therefore, even though the heterogeneous semantics, libraries and platform support make it very challenging, portability is an important concern in the use of DSP design tools. Note that portability of DSP designs across DSP design tools, when it is comprehensively supported, is equivalent to portability across all underlying embedded processing platforms and DSP code libraries supported by them.

Our initial efforts to address these goals focused on transferring dataflow technology across design tools. A critical issue arises here due to the lack of a standard and vendor-independent language, and associated intermediate representations together with efficient implementations of algorithms for operating on these representations. The dataflow interchange format (DIF) and the associated DIF package, a Java package that provides dataflow-based representations and algorithm implementations, have been developed for these purposes of specifying and working with DSP applications across the evolving family of dataflow-based design tools.

In this paper, we introduce our further progress with the DIF language and the DIF package for porting DSP applications across dataflow-based design tools. Such porting typically requires tedious effort and is highly error-prone. This portability can be a powerful capability if it is attained through a high degree of automation, and a correspondingly low level of manual or otherwise ad-hoc fine-tuning. This motivates a new porting approach that we have developed through the dataflow information captured by the DIF language, and through additional infrastructure for converting dataflow-based application models to and from DIF, as well as for mapping tool-specific actors based on the information specified by the actor interchange format.

This paper presents a summary of the new porting capabilities in DIF. For more complete development of these capabilities, we refer the reader to [6].

2. The Dataflow Interchange Format

The dataflow interchange format [5] is a language for specifying mixed-grain dataflow models for digital signal processing (DSP) systems and other streaming-related application domains. It provides designers a unique, integrated set of semantic features that are relevant to dataflow modeling and dataflow-based DSP application programming. Specifically, it is designed to describe graph topologies and hierarchies as well as to specify dataflow-related

and actor-specific information. The dataflow semantics of a DSP application have a common representation in DIF regardless of the particular tool used originally to enter the application specification. DSP applications specified by the DIF language are referred to as *DIF specifications*.

Interchanging information between general software tools through meta-modeling has been investigated by several works, e.g., CDIF [3] and MOF [9]. In contrast, DIF is specifically designed for the emerging dataflow-based DSP prototyping tools. DIF is not based on XML, which is more for pure data exchange applications and is not particularly suited for programming dataflow-based applications.

2.1. Dataflow Modeling

In the dataflow modeling paradigm, computational behavior is depicted as a dataflow graph. A dataflow graph consists of a set of nodes and a set of directed edges. A node (actor) represents either an indivisible computation or a hierarchically-nested subgraph. An edge represents a FIFO queue that buffers tokens from its source to its sink. Dataflow graphs naturally capture the data-driven property that is inherent in most DSP computations. An actor can fire when it has sufficient tokens on all of its incoming edges. When firing, it consumes certain numbers of tokens from its incoming edges, executes the computation, and produces certain numbers of tokens on its outgoing edges.

Many dataflow models have been developed by various researchers, e.g., synchronous dataflow (SDF), cyclo-static dataflow (CSDF), homogeneous synchronous dataflow (HSDF), interval-rate locally-static dataflow (ILDF), Boolean dataflow (BDF), parameterized dataflow, and blocked dataflow (BLDF). See [6] for further discussion of such models as well as references to the original works. The current version of the DIF language, v0.2, has demonstrated its capability of specifying all of the aforesaid models [6].

```
dataflowModel graphID {
  basedon { graphID; }
  topology {
    nodes = nodeID, ...;
    edges = edgeID (srcNodeID, snkNodeID), ...; }
  interface {
    inputs = portID [:nodeID], ...;
    outputs = portID [:nodeID], ...; }
  parameter {
    paramID;
    paramID = value;
    paramID : range; }
  refinement {
    subgraphID = supernodeID;
    subPortID : edgeID;
    subParamID = paramID; }
  builtInAttr {
    [elementID] = value;
    [elementID] = id;
    [elementID] = id1, id2, ...; }
  attribute usrDefAttr{
    [elementID] = value;
    [elementID] = id;
    [elementID] = id1, id2, ...; }
  actor nodeID {
    computation = stringValue;
    attrID [:attrType] = value;
    attrID [:attrType] = id;
    attrID [:attrType] = id1, id2, ...; }
}
```

Figure 1. The DIF language version 0.2 syntax.

2.2. Dataflow Interchange Format Version 0.2

The overall language syntax of DIF v0.2 [6] is sketched in Figure 1. Items in boldface are built-in keywords; non-bold items are specified by users or generated by exporting tools; and items enclosed by squares are optional. Briefly speaking, the *dataflowModel* keyword specifies the dataflow modeling semantics of the graph. The *basedon* block provides a convenient way to refer to a pre-defined graph. The *topology* block specifies the nodes and edges of the graph. The *interface* block defines the input and output ports of the enclosing hierarchy. The *parameter* block is designed for specifying parameterized values, intervals or other ranges for such values, and value-unspecified attributes. The *refinement* block constructs hierarchical graph structures, provides details for interface connections, and sets subgraph parameters. The built-in attribute (*builtInAttr*) block specifies dataflow modeling information specific to the model used, and the user-defined attribute (*attribute*) block specifies user-defined attributes. The *actor* block specifies tool-specific actor information. More detailed description of the DIF language, including the language grammar, can be found in [6].

3. The DIF Package

The DIF package is a Java-based 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.

3.1. The DIF Representation

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

Figure 2 presents the class hierarchy of graph classes in

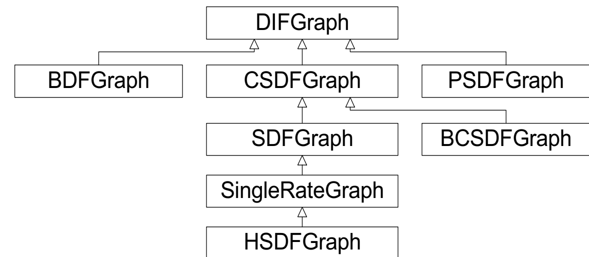


Figure 2. Dataflow graph classes in the DIF package.

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

3.2. The DIF Front-end and Implementations of Dataflow-based Algorithms

The DIF front-end tool provides users an integrated set of interfaces to convert between DIF specifications and the corresponding DIF representations. Specifically, the DIF language parser is implemented using a Java-based compiler-compiler called SableCC [4].

For supported dataflow models, the DIF package also provides efficient implementations of various scheduling and optimization algorithms that operate on DIF representations. These implementations provide designers a convenient interface to analyze and optimize DSP applications. It is also possible to integrate DSP design tools with the DIF package and then utilize the powerful representations, algorithm implementations, and other features provided by the DIF package.

3.3. The Methodology of Using DIF

Figure 3 illustrates the end-user viewpoint of the DIF architecture. DIF supports a layered design methodology covering dataflow models, the DIF package, dataflow-based DSP design and prototyping tools, and the underlying embedded processing platforms targeted by these tools.

The dataflow models layer represents the dataflow models currently integrated in the DIF package. 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 [2], developed at UC Berkeley, and the Autocoding Toolset [10] developed by Management Communications and Control, Inc. (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. Tools such as these form a layer in our proposed DIF-based design methodology. The embedded processing platforms layer gives examples of platforms supported by Ptolemy II and the Autocoding Toolset. In general, this layer represents all embedded platforms that are supported by dataflow-based DSP design tools.

The DIF package acts as an intermediate layer between abstract dataflow models and different practical implementations. DIF exporting and importing tools automate the process of translating between tool's specification formats and DIF specifications or DIF representations and provide a useful front-end to use DIF and the DIF package.

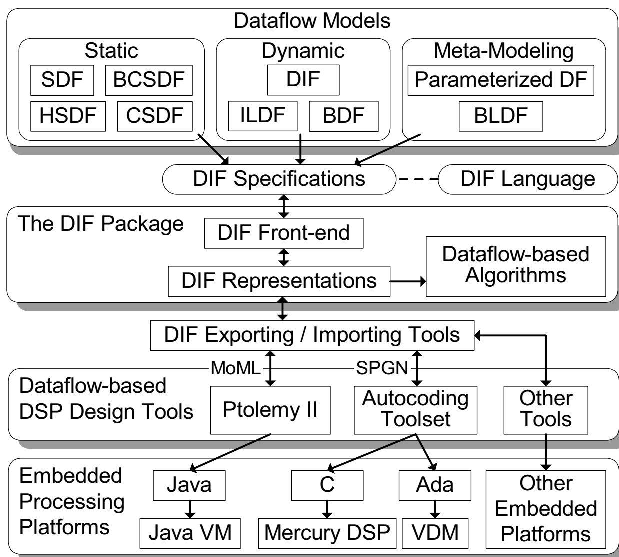


Figure 3. The role of DIF in DSP system design.

4. Exporting and Importing DIF

In DIF terminology, *exporting* means translating a DSP application from a tool’s specification format to DIF (either to the DIF language or directly to the appropriate form of DIF representation). On the other hand, *importing* means translating a DIF specification to a design tool’s specification format or converting a DIF representation to a tool's internal representations.

4.1. Mapping Dataflow Graphs

When exporting, parsing a tool’s specification format and then directly formulating the corresponding DIF specification is usually not the most efficient approach. Since DIF provides a complete set of classes for representing dataflow graphs through a well-designed, object-oriented realization, mapping the graphical (internal) representations of tools to the formal dataflow representations used in DIF, and then converting to DIF specifications is typically more convenient to develop and more efficient to execute.

We categorize implementation issues involved in this process as *dataflow graph mapping* issues. Dataflow-based design tools usually have their own representations instead of just the abstract components defined in theoretical dataflow models. Although these representations are tool-specific, exporting without losing any essential modeling

```
actor nodeID {
  computation = “ptolemy.domains.sdf.lib.FFT”;
  order : PARAMETER = intValue or intParamID;
  input : INPUT = incomingEdgeID;
  output : OUTPUT = outgoingEdgeID;
}
```

Figure 4. DIF actor specification of the FFT actor.

information is completely feasible due to the following properties of DIF. First, the DIF language is capable of describing dataflow semantics regardless of any particular tool used 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 properties, our general approach for exporting is to comprehensively traverse graphical representations in a design tool and then map the modeling components encountered into equivalent corresponding components or groups of components available in DIF representations.

4.2. Specifying Actors

Specifying an actor's computation and all necessary operational information is referred to as *actor specification*. Although this detailed information is not directly used by many dataflow-based analyses, it is essential in exporting, importing, and porting across tools, as well as in hardware/software synthesis since every actor's functionality must be fully preserved. The actor block in the DIF language v0.2 is designed for the actor specification [6].

Lets take the FFT operation as an example to illustrate actor specification in DIF. In Ptolemy II, the FFT actor is referred to as *ptolemy.domains.sdf.lib.FFT*, and it has a parameter *order* and two ports, *input* and *output*. The built-in DIF attribute *computation* and built-in attribute types *PARAMETER*, *INPUT*, and *OUTPUT* are used to specify its computation, parameters, and connections. The corresponding DIF actor specification is presented in Figure 4.

4.3. Exporting and Importing Mechanism

Figure 5 illustrates the exporting and importing mechanisms in DIF. First, a dataflow graph mapping algorithm must be properly designed for the specific design tool. Then a DIF exporter is implemented based on the graph mapping algorithm in order to convert the graphical representation in that design tool to an equivalent 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, based on a "reverse graph mapping algorithm" and actor specification, the DIF importer can construct the graphical representation in the tool while preserving the same functionality of the original application.

We have implemented a DIF exporter and a DIF importer for Ptolemy II [6]. 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 [8] specification with all functionality preserved. Such an equivalent result from round-trip translation validates the correctness of the strategies and general methods in DIF for dataflow graph mapping and actor specification.

5. Porting Mechanism

Figure 6 illustrates our newly developed 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 to introduce the porting mechanism in more detail.

The first step is to export a DSP application developed in the Autocoding Toolset (AT), which uses MCCI's Signal Processing Graph Notation (SPGN) as its specification format, to the corresponding DIF specification. In this stage, the actor information (actor specification 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 the Autocoding Toolset to Ptolemy II. 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 to an equivalent Ptolemy II Modeling Markup Language MoML [8] specification. This importing process is handled

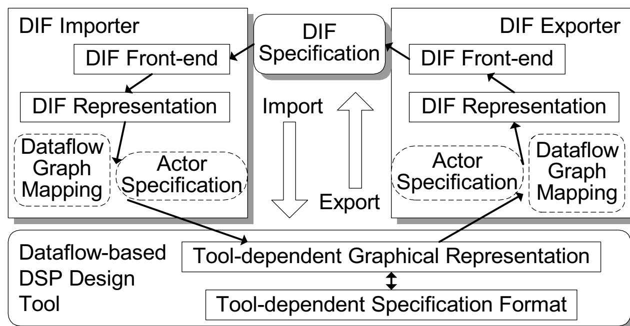


Figure 5. The exporting and importing mechanisms.

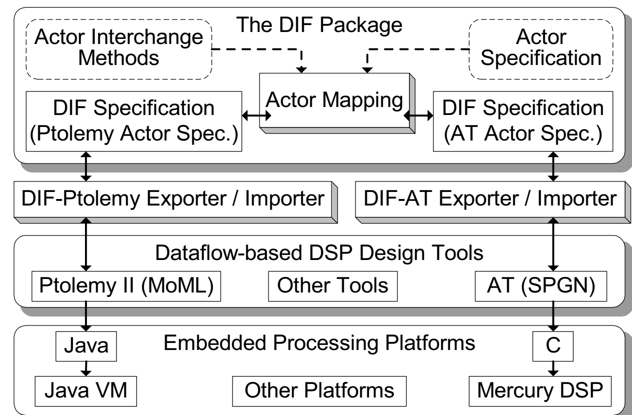


Figure 6. The DIF porting mechanism.

by the DIF-Ptolemy exporter/importer automatically.

The key advantage of using DIF as an intermediate state in the porting process is that except for the actor block, a DIF specification for a DSP application represents the same semantic information regardless of which design tool is used. Such unique dataflow 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.

5.1. Actor Mapping

The objective of *actor mapping* is to map an actor in a design tool to an actor or set of actors in another design tool while preserving the same functionality. Because different design tools generally 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, the *actor absence* problem arises. If corresponding actors exist in both libraries but the specific functionalities of those actors do not completely match, we encounter the *actor mismatch* problem. For example, the FFT domain primitive in the Autocoding Toolset allows users to indicate an FFT or IFFT operation through its parameter *FI*, but the FFT actor in Ptolemy II does not. *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.

The *actor interchange format* can significantly ease the burden of actor mismatch problems by allowing designers to specify how multiple actors in the target design tool can construct a subgraph such that the subgraph's functionality is compatible with the source actor. Such conversions reduce the need for users to introduce new actor definitions in the target tool. 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, DIF's actor mapping mechanism can take over the job efficiently and systematically.

5.2. Actor Interchange Format

The actor interchange format (AIF) is a specification format dedicated to specifying actor interchange information. It consists of the *actor-to-actor mapping* block and the *actor-to-subgraph mapping* block.

The actor-to-actor mapping block, as presented in Figure 7, specifies the mapping information from a source actor to a target actor. The *srcActor* and *trgActor* terms designate the computations of the source and target actors, respectively. A method *methodID* is given optionally to specify a prior condition that must be satisfied to trigger the mapping. The statements inside braces provide ways to specify or to map the target attribute values. First, AIF allows users to directly assign a value *value* for a target attribute *trgAtID*. Second, the value of *trgAtID* can be directly assigned by the value of a source attribute *srcAtID* if *methodID* is not given in this statement. 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.

The actor-to-subgraph mapping block, as presented in Figure 8, specifies the mapping from a source actor to a subgraph consisting of a set of target actors and depicts the topology and interface of this subgraph. It is designed for use when matching to a standalone actor in the target tool is not possible. The *trgGraph* term specifies the computation in order to invoke a subgraph component in the target tool. The *topology* block portrays the topology of *trgGraph* and the *interface* block defines the input and output ports of *trgGraph*. Mappings from the interface attributes of the *srcActor* to the interface ports of the *trgGraph* are also specified. The actor information of nodes in *trgGraph* is specified in *actor* blocks. More detailed description of the AIF, including the AIF grammar, can be found in [6].

5.3. Actor Interchange Methods

The methods optionally given in AIF specifications are used to perform conditional checks or to evaluate attribute values. They are referred to as *actor interchange methods*.

```
actor trgActor <- srcActor | methodID(arg, ...) {
  trgAtID = value;
  trgAtID <- srcAtID | methodID(arg, ...);
  trgAtID1, ..., trgAtIDn <- srcAtID;
  trgAtID <- srcAtID1, ..., srcAtIDn;
}
```

Figure 7. The AIF actor-to-actor mapping syntax.

```
graph trgGraph <- srcActor | methodID(arg, ...) {
  topology {
    nodes = nodeID, ...;
    edges = edgeID (srcNodeID, snkNodeID), ...; }
  interface {
    inputs = portID : nodeID <- srcAtID, ...;
    outputs = portID : nodeID <- srcAtID, ..., }
  actor nodeID {
    computation = "stringDescription";
    trgAtID = value;
    trgAtID = ID;
    trgAtID = ID1, ..., IDn;
    trgAtID <- srcAtID | methodID(arg, ...);
    trgAtID <- srcAtID1, ..., srcAtIDn; }
}
```

Figure 8. The AIF actor-to-subgraph mapping syntax.

A set of commonly-used interchange methods is defined in a built-in Java class in the DIF package. Users can extend this class and design specific interchange methods for more complicated or specialized actor mapping scenarios.

There are three built-in actor interchange methods in the DIF package: 1. *ifExpression (“expression”)* evaluates the Boolean *expression* and returns true or false; 2. *assign (“expression”)* evaluates the *expression* and returns the evaluated value; and 3. *conditionalAssign (“valueExpression”, “conditionalExpression”)* 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.

5.4. Case Study: FFT

According to the actor mismatch and attribute mismatch problems described in Section 5.1, the Autocoding Toolset FFT library module (referred to as D_FFT) can be mapped to the Ptolemy FFT actor only when its parameter *FI* is not set to indicate IFFT operation. Moreover, the parameter N of the Autocoding Toolset’s FFT module can be mapped to the parameter *order* of Ptolemy’s FFT actor only when $N = 2^{order}$ is satisfied, where N and *order* are integers. The AIF specification for mapping the FFT operation from the Autocoding Toolset to Ptolemy II is shown in Figure 9.

The D_FFT module also has a parameter B, which specifies the first point of its output sequence, and a parameter

```
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 9. AIF specification for mapping FFT.

```
graph ptolemy.actor.TypedCompositeActor<-D_FFT
   |ifExpression("FI==1 && M!=N") {
  topology {
    nodes = IFFT, Scale, SequenceToArray,
             ArrayExtract,ArrayToSequence;
    edges = ...; }
  interface { inputs = in:IFFT<-X;
    outputs = out:ArrayToSequence<-Y; }
  actor IFFT {
    computation="ptolemy.domains.sdf.lib.IFFT";
    order:PARAMETER <- N|conditionalAssign(..);
    ... }
  actor Scale { ...
    factor:PARAMETER <- N;
    ... }
  actor ArrayExtract { ...
    sourcePosition:PARAMETER<-B|assign("B-1");
    extractLength:PARAMETER <- M;
    ... }
  ...
}
```

Figure 10. AIF specification for mapping IFFT.

M, which specifies the number of output points. Furthermore, there is a factor of N difference between the IFFT operation of the D_FFT and the Ptolemy IFFT actor. One way to solve this problem is to create a new IFFT actor in Ptolemy, but this approach is relatively time-consuming. The actor-to-subgraph mapping feature in DIF can be used as a more convenient alternative. Figure 10 presents the critical part of an AIF specification that achieves this. If a D_FFT domain primitive indicates an IFFT operation (*FI == 1*) and it outputs only part of its sequence ($M \neq N$), it is mapped to a Ptolemy subgraph consisting of an IFFT actor for performing an IFFT operation, a Scale actor for adjusting each sample by a factor of N, and three array processing actors for extracting a certain part of the output sequence. The input and output ports of the subgraph, *in* and *out*, are mapped from parameters X and Y of D_FFT.

6. Experiment of Porting SAR Benchmark

In this section, we port a synthetic aperture radar (SAR) benchmark application from the MCCI Autocoding Toolset to Ptolemy II and in doing so, we demonstrate the effectiveness of our newly-developed porting mechanisms in DIF. The synthetic aperture radar system examined here was used as a benchmark in the Rapid Prototyping of Application Specific Signal Processors (RASSP) program sponsored by DARPA [10].

Figure 11 shows the SAR system developed by MCCI using the Autocoding Toolset [10]. Figure 11(a) illustrates the top-level dataflow graph. This graph consists of two major building blocks, *RANGE* processing and *AZIMUTH* processing, represented respectively by the RNG subgraph in Figure 11(b) and the AZI subgraph in Figure 11(c).

With a properly-designed actor interchange specification [6] together with actor interchange methods available in the DIF package, the DIF actor mapping mechanism can translate the DIF specification of Figure 11, which is

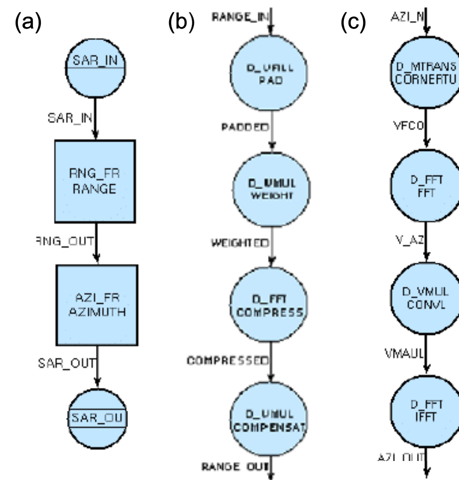


Figure 11. The SAR system in the Autocoding Toolset.

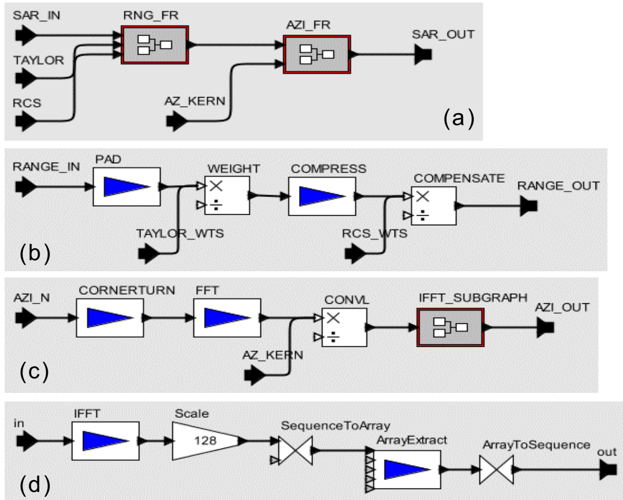


Figure 12. The ported SAR system in Ptolemy II.

exported from the Autocoding Toolset, to an equivalent DIF specification for Ptolemy II. The DIF-Ptolemy exporter/importer can then import this equivalent specification; the resulting graphical representation in Ptolemy II is shown in Figure 12. Figure 12(a), 12(b), and 12(c) correspond to Figure 11(a), 11(b), and 11(c), respectively. Note that the node IFFT in Figure 11(c) is mapped to the IFFT_SUBGRAPH in Figure 12(d) through the AIF actor-to-subgraph mapping capability.

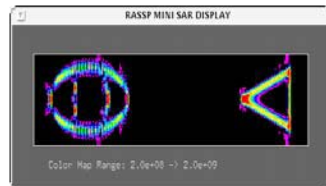
The ported SAR benchmark application in Ptolemy II works correctly. Figure 13 compares the output samples generated by Ptolemy II with those generated by the Autocoding Toolset, and reveals that the simulation results are the same except for tolerable precision errors.

7. Summary and Current Status

In this paper, we have reviewed the principles behind the dataflow interchange format (DIF) and the DIF package. We have then described our new approach to automate the processes of exporting and importing DIF when interfacing to DSP design tools. Finally, we have developed the DIF porting mechanism, and demonstrated it through experiments using a synthetic aperture radar system.

Ptolemy II

```
1.113328370E9, -5.672582199E8
1.686243152E9, -1.132239286E9
2.280892492E9, -1.837179778E9
2.787030647E9, -2.565079199E9
3.121469726E9, -3.124321013E9
3.235633491E9, -3.339997173E9
3.126105298E9, -3.132702116E9
2.795907223E9, -2.578937710E9
2.292518065E9, -1.852489499E9
1.698661416E9, -1.145532955E9
```



MCCI

```
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
```

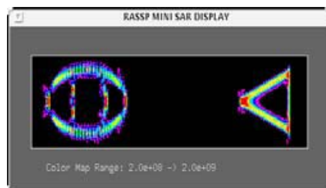


Figure 13. Simulation results of the SAR systems in Ptolemy II and the MCCI Autocoding Toolset.

Our ongoing work on DIF includes using AIF to develop a component within the DIF package to interface DIF to and from VSIPL [7], which is an important publicly-available library for DSP software design. Such a capability would enable tools to port designs efficiently to VSIPL-based implementations, and to port designs to other tools through the AIF-based integration of DIF and VSIPL.

DIF is being developed in the University of Maryland DSP-CAD Research Group. Currently, DIF is being evaluated and used by a number of research partners, including MCCI, which has developed DIF exporting and importing capabilities in its Autocoding Toolset. A general public release of DIF is being planned for the near future.

8. Acknowledgements

This research was supported by the U. S. Defense Advanced Research Projects Agency (DARPA) via the U. S. Army Aviation and Missile Command (Contract Number DAAH01-03-C-R236).

References

[1] 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.

[2] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming Heterogeneity - the Ptolemy Approach," *Proceedings of the IEEE*, v.91, No. 2, January 2003.

[3] R. Flatscher, "Metamodeling in EIA/CDIF - Meta-Metamodel and Metamodels", *ACM Transactions on Modeling and Computer Simulation*, vol. 12, no.4, pp. 322-342, October 2002.

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

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

[6] C. Hsu and S. S. Bhattacharyya, "Dataflow Interchange Format Version 0.2", Technical Report UMIACS-TR-2004-66, Institute for Advanced Computer Studies, University of Maryland, College Park, November 2004.

[7] R. Janka, R. Judd, J. Lebak, M. Richards, and D. Campbell, "VSIPL: An object-based open standard API for vector, signal, and image processing," in *Proc. 2001 IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, vol. 2, pp. 949–952.

[8] E. A. Lee and S. Neuendorffer, "MoML - A Modeling Markup Language in XML, Version 0.4", Technical Memorandum UCB/ERL M00/12, Univ. of California, Berkeley, March 2000.

[9] Object Management Group, *Meta Object Facility (MOF) Specification*, v. 1.4, April 2002. In http://www.omg.org.

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