In Proceedings of the IEEE Workshop on Signal Processing Systems, pages 10-15, Banff, Canada, October 2006.

Configuration and Representation of Large-Scale Dataflow Graphs using the Dataflow Interchange Format

Ivan Corretjer, Chia-Jui Hsu, and Shuvra S. Bhattacharyya, *Member, IEEE*

***Abstract*—A wide variety of DSP design tools have been developed that incorporate dataflow graph representations into their GUI-based design environments. However, as the complexity of application graph topologies increases, textual manipulation of graph specifications becomes increasingly important. The dataflow interchange format (DIF) provides a text-based language for the description of dataflow graphs. Currently, the DIF infrastructure supports the specification of mixed-grain dataflow models, porting of dataflow applications specified in DIF across DSP design tools, software synthesis of applications specified in DIF, as well as a variety of optimization and analysis capabilities. This paper presents a novel set of dataflow graph configuration features that have been developed in the DIF language. These features greatly enhance the flexibility and power with which dataflow graphs, especially large-scale graphs, can be constructed and manipulated in DIF. To support the new graph configuration capabilities, several new concepts have been incorporated into the DIF language semantics, such as the capability to handle certain dynamic dataflow constructs, and support for C-like arrays in DIF specifications. Along with these concepts, a new framework for the construction and manipulation of DIF objects through the use of C/C++ is presented, and applications of this framework are demonstrated.**

I. Introduction

THIS paper presents a new framework for construction and manipulation of dataflow graphs for DSP system design. To facilitate design of complex systems, our framework includes a powerful integration of C-based procedural programming with dataflow graph construction primitives. The C programming language and dataflow graphs have been evolving as complementary, de facto standards for the high-level specification and design of digital signal processing systems [3]. C provides for high-level programming constructs while also providing designers with a relatively large degree of control over the efficiency of software implementation. Dataflow, on the other hand, provides for formal properties and exposes coarse-grained design structure that is valuable for high level optimization of embedded software.

Dataflow is widely used for designing DSP applications, and has a rich history with numerous computational models [2, 4, 5, 8, 15, 17, 18] and design tools [6, 7, 10, 14, 20, 21, 22] developed over the years that directly support or incorporate dataflow semantics. In contrast to C's textual representation of DSP applications, dataflow models provide an intuitive graphical representation of an application to designers. However, as the complexity of an application increases textual manipulation of graphs becomes increasingly important. For example, simple low-level graphs may be developed by designers in GUI based tools; however, the combination of many simpler modules into complex graphs and subgraph hierarchies often lends itself more toward specification through textual constructs, and automated manipulation by computer programs.

In the context that we discuss dataflow in this paper, dataflow is a programming model in which a signal processing application program is represented as a directed graph. Vertices in this graph, called *actors*, correspond to functional modules. Edges specify the flow of data between modules, and correspond to first-in-first-out (FIFO) buffers. Actors can be of arbitrary complexity. Examples of actors that are commonly employed in dataflow-based design tools range from simple "atomic" operations such as addition and subtraction to more complex operations such as FIR and IIR filters, and FFT computations.

An actor in a dataflow graph can execute whenever it has sufficient data on its input edges. When an actor executes it consumes some amount of data from its input edges and produces some amount of data on its output edges. When analyzing dataflow graphs for verification or efficient synthesis, it is often useful to develop characterizations of the rate at which each actor produces and consumes data with respect to each of its incident edges. For example, such a production or consumption "rate" characterization can be a constant value, as in synchronous dataflow (SDF) [15]; a multidimensional volume of fixed dimensions, as in multidimensional synchronous dataflow [17]; a periodic sequence of constant values, as in cyclo-static dataflow [4]; a dynamically-parameterized expression, as in parameterized synchronous dataflow [2]; a bounded unknown value, as in bounded dynamic dataflow [18]; or a control-dependent value, as supported in different ways in integer-controlled dataflow [5] and Cal [7].

An important goal in the DIF language is to carefully

Manuscript received May 8, 2006. This work was supported in part by the U.S. Defense Advanced Research Projects Agency through Management, Communications, and Control, Inc.

Ivan Corretjer is with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742 USA (phone: 301-405-3744; e-mail: icorretj@eng.umd.edu).

Chia-Jui Hsu is with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742 USA (e-mail: jerryhsu@eng.umd.edu)

Shuvra S. Bhattacharyya is with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742 USA (e-mail: ssb@eng.umd.edu)

support specification of and analysis using dataflow production and consumption rate characterizations. This is one of the most important and distinctive considerations involved in the system-level analysis of dataflow representations for signal processing applications [1].

II. DIF OVERVIEW

The dataflow interchange format (DIF) project is an effort undertaken in the DSPCAD Research Group at the University of Maryland to standardize dataflow semantics, facilitate technology transfer for DSP design tools, and improve dataflow modeling and synthesis technology. DIF attempts to unify important forms of DSP-oriented dataflow semantics into a single language, and also provides various components for the manipulation of graphs that are described in DIF. Figure 1 illustrates the role of DIF in DSP system design. Currently, the DIF design tool supports the specification of mixed-grain dataflow models [13], porting of dataflow applications specified in DIF across DSP design tools [11], software synthesis of applications specified in DIF [12], and a variety of dataflow-based optimization and analysis capabilities.

Like Silage [10] and StreamIt [22], DIF is a textual, DSP-oriented language. However, DIF is different from Silage and StreamIt in its emphasis on supporting and unifying a broad range of different dataflow modeling and meta-modeling styles, and its associated emphasis on supporting high level application analyses, such as analyses of interactions among dataflow production and consumption rates, scheduling, memory requirements, and performance.

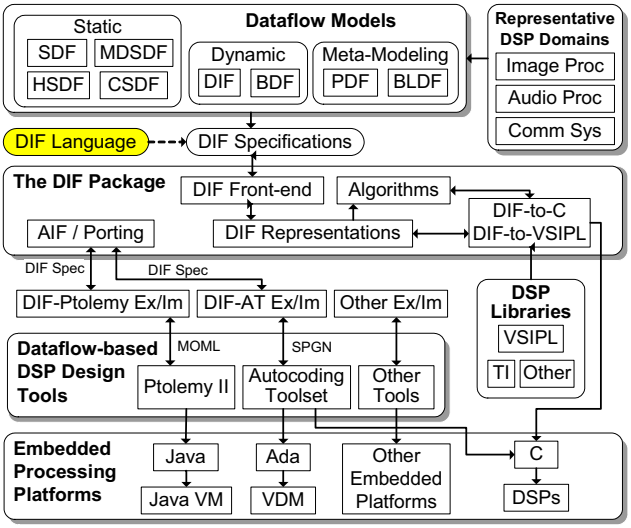

Fig. 1. An illustration of the role of DIF in DSP design.

This paper proposes a new set of features for DIF that greatly enhance the flexibility and power with which graphs, especially large-scale graphs, can be constructed and manipulated in DIF. Several new concepts have been incorporated into the DIF language semantics and are introduced in this paper, such as the capability to handle certain dynamic dataflow constructs, and support for C-like arrays in DIF specifications. Along with these concepts, a new framework for the construction and manipulation of DIF representations through the use of C/C++ is presented, and applications of this framework are demonstrated.

III. DIF LANGUAGE ENHANCEMENTS

By *configuration* of a dataflow graph, we mean the process of defining all relevant functional properties of the graph. The definitions involved in this process could be static (fixed for the entire execution of the graph) or dynamic (fixed only temporarily and subject to change as the graph continues executing). In this section, we introduce dataflow constructs that we have recently added to the DIF language to support more flexible and scalable graph configuration. These constructs address matters such as supporting dynamic reconfiguration aspects of dataflow designs, and providing facilities to more easily handle large-scale graphs through the integration of array notation into DIF-based graph construction.

A. Expressions and Parameters

DIF has been enhanced with support for sophisticated expressions for edge attributes, specifically for dataflow production and consumption rates. For certain dataflow models, such as SDF, these rates are constant integer values. For other dataflow models, such as Boolean dataflow [5] or parameterized dataflow [2], these rates can be specified in terms of symbolic parameters. In DIF, support for expressions in edge attributes provides a mechanism to specify actors whose production and consumption rates vary by arbitrary expressions. Arbitrary attribute expressions are supported in DIF, with the only constraint that the associated dataflow model support expressions as part of its specification format, and the resulting assignment, as a result of evaluating the expression, is a valid assignment of the edge attribute under the current dataflow models semantics. For models that support dynamic changes to parameter values, the latter condition may require run-time checking to fully enforce.

B. External Interfaces

When implemented or simulated, a dataflow graph specified in DIF typically interfaces with the external environment in one or more ways. A new feature added to DIF is support for end-user and device-related interfaces to graphs specified by DIF. Our present experimentation with this support has focused on GUI-based applications, e.g. as encountered in simulation tools; however, the underlying features are amenable to other kinds of interfacing contexts as well. We support external interfaces in DIF through the new *built-in* graph attribute `external`. Attributes of modeling objects in DIF can be *user-defined* or *built-in*. User-defined attributes provide a general, extensible way for users and tool developers to add application- and tool-specific information to a design. On the other hand, built-in attributes are attributes that are supported directly in DIF, through language keywords, and specialized data structures,

analysis mechanisms, or synthesis capabilities. The syntax for the new `external` attribute specifies an identifier for the external element and its assignment to an internal DIF modeling object (e.g., actor, edge, port, or interface). The following code fragment in DIF illustrates use of the `external` attribute.

```
attribute external {
   controlKnob1: interface1;
   controlKnob2: param1;
   display1: port2;
}
```

Use of the `external` attribute builds upon the existing interface/port syntax in DIF for connecting DIF objects in hierarchical and non-hierarchical configurations. Here, the external control knobs and the display are permanently bound to specific DIF graph objects. However, unlike interfaces and ports, connections made with the `external` attribute prevent the graph from being embedded into another DIF graph. That is, the assignment of an external attribute to a DIF object explicitly makes the current graph a top-level graph. This restriction is natural because the external attribute is intended only for connections that are not related to the underlying dataflow model but external to it.

C. *Arrays of DIF Objects*

The enhancements to DIF described in Sections III-A and III-B provide new mechanisms to interface DIF specifications with specific tools and applications, and leave significant aspects of the interpretation of the supported features to the end use contexts. In this section, we introduce a new feature of DIF that is supported by DIF internally, namely the integration of dataflow models with arrays, which are widely-used constructs in textual, procedural languages such as C.

The C programming language provides the capability to declare arrays of variables. Arrays provide a convenient organization for uniformly-typed data items. We adopt arrays in DIF following similar declaration syntax, but with somewhat different semantics. Whereas in C, and related languages, arrays are used primarily as storage constructs, arrays in DIF have different applications. First, arrays of graph *nodes* (dataflow actors) and edges can be used to configure a dataflow graph with sets of related modeling objects, and regular interconnections between such related objects can be made through concise syntax. Second, DIF operations can transform arrays and manipulate their attributes. Thus, DIF arrays serve as more than a convenient data structure, they represent actual objects in a dataflow graph and our semantics capture this difference. The following examples illustrate construction of graphs using arrays in DIF.

```
//Mixed array and singleton actors.
nodes= node1, node2[2], node3[4], node4;
```

As defined above, an array in DIF is actually a shorthand notation for a group of individual objects. Internally, DIF expands array declarations using a *baseName_index* naming convention. Note that any valid DIF *baseName* can be used in an array declaration, and that the *_index* suffix is always appended to the end of the *baseName*. Once a *baseName* is used in an array declaration neither it nor its expanded names can be reused in the current namespace. These two properties ensure that whether an array is referenced by *baseName* only or by the expanded *baseName_index* that each refers to a single object. The general naming scheme used by DIF is thus:

```
baseName[i] →
   baseName_1, baseName_2,…, baseName_i;
```

The array syntax also applies to any DIF modeling object (graph, parameter, attribute, etc.). In each case, the array can be thought of as declaring a group of individual instances of the DIF object. To further the applicability of arrays in DIF, the special case of assigning arrays of nodes to edges is treated next.

The differences in semantics between DIF arrays and C arrays become apparent in the case of assigning arrays of nodes to edges. In addition to the expanded naming convention defined above for arrays, we support the same expanded naming convention for edges created implicitly by connecting arrays of nodes together:

```
edges = e1(node1, node2); →
   e1_1(node1, node2_1),
   e1_2(node1, node2_2);
```

As seen by the above example, even though edge *e1* is not explicitly declared as an array, since its sink node is an array, two copies of *e1* are created using the array naming convention. This extra functionality of implicit arrays allows for a multitude of assignment forms. For example, it is possible to have an edge whose source node is an array of 2 nodes and whose sink node is an array of 4 nodes. In this case it is not clear what the graph topology should look like. To handle these ambiguities, we first restrict the implicit creation of arrays to certain cases involving the use of arrays of nodes and edges (for other DIF objects, arrays are strictly shorthand notation and each object should be treated and referenced individually). We then extend the array syntax described above for general DIF objects to deal with more elaborate node/edge interconnection in Section IV-A. The following code fragment illustrates the restrictive cases.

```
nodes = a[N], b[M], c;
edges = d(a,b), e(b,c);
```

From this syntax, we identify two cases: 1) both source and sink nodes are arrays, and 2) either the source or sink

node is an array, and the corresponding sink or source node is a singleton. For the first case we require that $N = M$, and then the result of defining edge d is to create the following set of edges:

```
{d_1(a_1, b_1), d_2(a_2, b_2),…,
      d_N(a_N, b_N)}.
```

For the second case — in which either a source or sink node array connects to a singleton — there is no restriction and the semantics is to create M edges. For the e array, each instance has the corresponding instance of b as its source and the singleton c as its sink. Recalling the naming convention described earlier, the result of the second case is the creation of the following set of edges:

```
{e_1(b_1, c),e_2(b_2, c),…,e_M(b_M, c)}.
```

Finally, we note that since the array syntax in DIF is short-hand for the expanded *baseName_index* naming scheme, the individual elements of arrays can be referenced directly in DIF specifications using the expanded nomenclature.

IV. Generalized configuration support

The new language features presented in the previous section generally extend the range of dataflow related concepts that can be represented by DIF and increase the expressiveness of DIF specifications. These features are oriented at handling individual modeling constructs and related groups of modeling constructs that are arranged in regular patterns (e.g., multiple edges originating from the same source node).

However, as we noted in introducing arrays, with the new expressiveness of DIF comes the need to be able to occasionally handle more general types of graph configuration structures and complex topologies.

A. C/C++ based Configuration Framework

To address this and other related issues we introduce a new framework for manipulating DIF representations that allows for "arbitrary" C/C++ code to be interfaced with DIF to configure and manipulate graphs. By leveraging the power of general-purpose procedural programming in a restricted way, this framework greatly extends the power of the DIF specification format. Our hybrid procedural/dataflow configuration framework allows users to define arbitrary modules of C/C++ code that access edge, node, and graph attributes, and use values of such attributes along with arbitrary control logic (if-else, for loops, etc.) and user input (e.g., from user-specified configuration files), to configure that graph. Our choice of C/C++ as the procedural programming language here is motivated by the popularity of C in the signal processing domain.

To enable such an integrated approach to dataflow graph construction, we have developed a C/C++ support library for DIF. This library interfaces to DIF via Java's Native Interface (JNI) [16] along with a wrapper generator system (JACE) [19]. The JNI allows for the interfacing of existing Java classes to C/C++ objects. The mechanism makes use of bindings between the two languages and maps Java data types to native C/C++ data types. In addition the JNI provides for the calling of Java member functions from C/C++. This low-level interface involves management of a scaled down Java virtual machine (JVM) from within C/C++, with the programmer having to manage references to objects and memory allocation/deallocation. To ease this burden on the programmer, we apply the JACE wrapper generator [19]. The resulting library internally handles the specifics of JNI interfacing with C/C++ and automatically generates C/C++ wrapper classes for Java classes. This mechanism provides full access to DIF functions.

However, to further the ease of using DIF in C/C++, we provide our own general framework, which presents a simplified interface to DIF. Our framework is targeted towards the configuration and manipulation of large-scale graphs and accomplishes this by wrapping complex sequences of native DIF calls into simple C/C++ functions. Using this combination of tools allows the C/C++ framework to present the DSP designer with familiar C/C++ functional access to DIF.

Finally, we note that an alternative to providing this C/C++ framework for manipulating DIF objects would be to incorporate procedural, control-flow constructs directly into the DIF language. We do not use this approach to avoid cluttering the syntax and implementation of DIF, make it easier to learn DIF (by using the familiar C language wherever appropriate), and avoid redundancy and duplication of features that are already available in C.

V. DIFdoc

Tools for dataflow-based modeling and design have traditionally been GUI based. These tools present to the designer a graphical representation (in the form of a graph) of the application currently under development. Two features that are not present in these design tools are the ability to examine an entire graph hierarchy from a single view, and the ability to document a dataflow design hierarchy in a standard format.

We have developed a tool-independent, human-readable dataflow graph documentation format called DIFdoc for representing dataflow designs as hyperlinked combinations of HTML files and visual, graph representations. We have also developed a tool for generating DIFdoc representations from the DIF internal representation, which is the internal form to which all DIF language specifications are compiled into by the DIF language front-end, and on which all dataflow analysis algorithms in the DIF software package operate.

More specifically, the DIFdoc format displays dataflow-based design hierarchies using a combination of HTML (to

represent hierarchical layers textually), and *dot* graphs (to represent individual graphs pictorially). The *dot* package is a well-known, freely-available software package for drawing graphs in their standard pictorial format [9].

The HTML portions of DIFdoc use indentation to represent the various layers of the design hierarchy, and hyperlinks so that for any level of the hierarchy, one has easy access to a visual (dot-based) representation for the dataflow graph at that level. Furthermore, deeply-nested designs can be organized through multiple HTML files, where the deepest levels of the hierarchy in the top-level representation are linked to separate HTML-based representations of their internal structures.

For example, consider a simple top-level dataflow graph *X*. In *X*, we have two actors *A* and *B* that are connected (e.g., by an edge *e* from *A* to *B*). Actor *A* is a hierarchical actor, and its associated subgraph (refinement) consists of two actors *C* and *D*.

This design hierarchy would then result in the HTML layout illustrated in Figure 2(a). Here *A* and *B* are shown to be in the top-level of the hierarchy, with *C* and *D* being embedded within *A*. Using the basic hyperlink mechanism in HTML, each hierarchical actor links to an associated pictorial graph layout that has been plotted in advance using the *dot* tool. Even the top-level graph *X* is linked to an associated pictorial representation (this contains a pictorial layout of *A*, *B* and *e*, as shown in Figure 2(b)).

This hyperlinked, combined textual-pictorial representation of dataflow designs provides a unique representation of an entire dataflow hierarchy, a kind of view that is not available with conventional GUI-based dataflow tools, and is an especially convenient way for users to browse dataflow designs, and document them in a tool-independent way.

VI. Application Examples

Benchmarks of our C/C++ configuration framework run against the native Java code show that the performance of DIF is not degraded significantly through use of the framework (in some cases the performance actually improves due to the enabling of more streamlined configuration constructs). Here, by the native Java code, we mean by explicit specification of all nodes, edges, and other

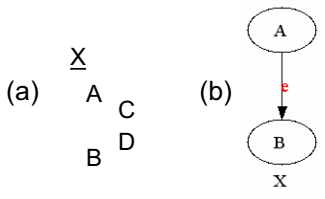

Fig. 2. A simple illustration of DIFdoc

modeling objects without using arrays, without using the integrated C/C++ configuration features, and therefore relying only on the underlying ("native") Java facilities that support the core implementation of the DIF package.

Table 1 shows the results of running several benchmarks through both the import and export functions of DIF as well as the newly introduced DIFdoc. The data was collected using the Unix *time* command, which returns the total execution time and the maximum number of memory pages allocated during a program's execution.

Table 1. Performance of native vs. framework executables.

Import and Export

		Native		Framework	
	filesize (kB)	time (us)	memory (kB)	time (us)	memory (kB)
graph4	5	0.0148	5488	0.0150	5520
graph1_8	4	0.0144	5488	0.0138	5536
MCCISAR	3	0.0098	5488	0.0100	5520
QAM16ED	12	0.0229	5488	0.0223	5536
BIG_FFT_EAG	29	0.1793	5488	0.1706	5536
FilterBank	41	0.0783	5488	0.0787	5536

HTML Generation

		Native		Framework	
	filesize (kB)	time (us)	memory (kB)	time (us)	memory (kB)
graph4	5	0.0159	5488	0.0147	5520
graph1_8	4	0.0164	5488	0.0155	5520
MCCISAR	3	0.0112	5488	0.0108	5520
QAM16ED	12	0.0286	5488	0.0279	5520
BIG_FFT_EAG	29	0.2098	5488	0.2008	5520
FilterBank	41	0.0951	5488	0.0978	5520

Here, graph4 and graph1_8 are synthetic benchmarks used to verify correctness of the framework and to establish baseline measurements. The MCCISAR and QAM16ED benchmarks exercise the hierarchical related functions of the import/export and DIFdoc generator codes while maintaining relatively simple topologies. The BIG_EAG benchmark is a parallel FFT benchmark. This benchmark contains the largest, most complex topology of all the benchmarks, although it does not contain any hierarchical actors. Finally, the FilterBank benchmark combines complex topology with hierarchical actors, exercising the full range capabilities within the DIF import/export and DIFdoc functions. Comparing the native and framework runtimes shows that on average, the native code and framework code take roughly the same amount of time to execute. The framework does have a slightly larger memory overhead, however, this overhead remains constant throughout each of the benchmarks. These results indicate that application of our integrated procedural-dataflow graph configuration framework can be performed without significant performance degradation.

To illustrate the benefit of DIF arrays and the C/C++ framework, the parallel FFT example, provided by MCCI [20], was chosen. This application encompasses a 6 stage parallel implementation of an FFT. Each stage is an identical FFT, but as the stages progress the interconnections among the various stages change (See Figure 3).

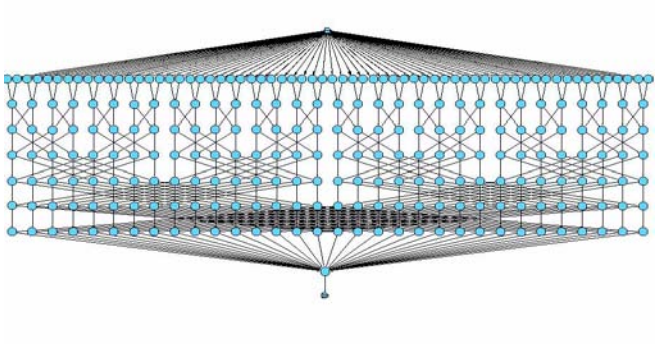

Fig. 3. Parallel FFT example.

Using the new array syntax, we can easily specify the FFT stages (e.g., `Stage1[32], Stage2[32], Stage3[32],` etc.). As previously discussed, this shorthand array notation expands out to iterate across all 32 processing elements per stage. To handle the interconnections among the various stages, the network was analyzed and a procedural for-loop was extracted for the main graph interconnections. Taking advantage of parameterization, this for-loop could be placed inside an outer loop to iterate over each of the 6 stages. After array expansion and application of the C/C++ framework, the resulting DIF specification included over 740 lines. This specification was generated using less than 120 lines of actual C/C++ code (including headers and preliminary setup code), with a further reduction to 70 lines if the outer loop optimization mentioned above is applied.

VII. Conclusions

In this paper, we have introduced several new concepts and features in the dataflow interchange format (DIF) that streamline support for large-scale graphs. Major contributions include the ability to specify and manipulate arrays of DIF objects, and to interface arbitrary C/C++ code to DIF to allow for powerful graph configuration and manipulation. The major enhancements described in this paper are implemented in prototype form, and are being incorporated into DIF version 1.0, which will be the first general public release of DIF. Currently, DIF is being evaluated and used by a number of industry- and university-based research partners.

References

[1] 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.

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

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

[5] J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 508-513, October 1994.

[6] J. Buck and R. Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the International Workshop on Hardware/Software Co-Design*, May 2000.

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

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

[9] E. R. Gansner, S. C. North, E. Koutsofios, and K.-Phong Vo. A technique for drawing directed graphs. Technical report, AT&T Bell Laboratories, 1993.

[10] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, and H. De Man. DSP specification using the silage language. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1990.

[11] C. Hsu and S. S. Bhattacharyya, "Porting DSP applications across design tools using the dataflow interchange format", *Proceedings of the International Workshop on Rapid System Prototyping*, Montreal, Canada, pp. 40-46, June 2005.

[12] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format", *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 37-49, Dallas, Texas, September 2005.

[13] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya, "DIF: An interchange format for dataflow-based design tools", *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 423-432, Samos, Greece, July 2004.

[14] Lauwereins, R., Engels, M., Ade, M., and Peperstraete, J. A, "Grape-II: A system-level prototyping environment for DSP applications", *IEEE Computer Magazine*, 28(2):35-43, February 1995.

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

[16] S. Liang. The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley, 1999.

[17] Murthy, P. K. and Lee, E. A, "Multidimensional synchronous dataflow", *IEEE Transactions on Signal Processing*, 50(8):2064-2079, August 2002.

[18] M. Pankert, O. Mauss, S. Ritz, and H. Meyr. Dynamic data flow and control flow in high level DSP code synthesis. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1994.

[19] T. Reyelts. Integrating Java and C++ with Jace. Javaworld.com, May 2002.

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

[21] T. Stefanov, et al. System design using Kahn process networks: the Compaan/Laura approach. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, February 2004.

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