June 16, 1994

# Software Synthesis for Single-Processor DSP Systems Using Ptolemy

Master's Report

José Luis Pino

Department of Electrical Engineering
and Computer Science
University of California
Berkeley, California 94720

## Abstract

Ptolemy is an environment for simulation, prototyping, and software synthesis for heterogeneous systems. It uses modern object-oriented software technology (in C++) to model each subsystem in a natural and efficient manner, and to integrate these subsystems into a whole. The objectives of Ptolemy encompass practically all aspects of designing signal processing and communications systems, ranging from algorithms and communication strategies, through simulation, hardware and software design, parallel computing, to generation of real-time prototypes. In this paper I will describe the software synthesis aspects of the Ptolemy system for single-processor architectures. The environment presented here is both modular and extensible.

# Acknowledgments

This paper is dedicated to my wife and children, with whose love and patience makes pursing a graduate education possible.

The work that led to this paper would not have been possible without the assistance of my advisor, Edward Lee, and the Ptolemy Team. In particular, I wish to thank Joseph Buck, Soonhoi Ha, Tom Parks, and Kennard White.

The author gratefully acknowledges the support of AT&T Bell Labs and Office of Naval Research.

# Table of Contents

# List of Figures

# 1.0 Introduction

Practical signal processing systems today are rarely implemented without software or firmware, even at the ASIC level. Programmable DSPs, in particular, form the heart of many implementations. An aggressive new implementation technology is to use one or more “DSP cores” together with custom circuitry. DSP cores are programmable architectures sold as silicon macro blocks rather than as separate components. They are used as large macrocells in application-specific ICs. Such ASICs are customized to contain precisely the memory and peripherals required by an application, and can also include arbitrary custom logic, configurable logic, or analog circuitry.

The first major market for DSP cores is digital cellular telephony. DSP vendors have developed specialized versions of their commodity DSPs that support both the GSM standard (for Europe) and the IS-54 standard (for the U.S.). For example, the Ericsson HotLine GH197 is a GSM hand-held telephone that uses an ADSP-2102 from Analog Devices. The Motorola DSP56156 is a DSP with carefully chosen peripherals and memory capacity to support the European GSM standard. The Motorola DSP56166 is a variant capable of implementing the VSELP speech coder in the U.S. and Japanese digital cellular standards.

So far, however, the customized core-based ASICs for this application are being designed by the DSP vendor, and not by the producer of the telephone equipment. This approach is viable because the functionality of the ASIC is specified by an international standard, and the market is expected to be very large. However, more proprietary designs cannot proceed in this manner. The design process will more closely resemble that of board-level products using commodity DSPs. Such designs, of course, are mixed hardware and software designs. Our approach to code generation is carefully architected to support such heterogeneous designs.

Any complete system design methodology, therefore, must include software synthesis for programmable devices. Mainstream design tool vendors for signal processing, such as those provided by Comdisco Systems, Mentor Graphics, and CADIS, have recognized this. They have all recently added software synthesis for DSPs to their tools (see for example [1] and [2]).

Looking forward, future tools should also include high-level software synthesis for real-time control as well as coupling to high-level hardware synthesis tools. Since the design styles for these capabilities are likely to be radically different from one another, the ideal methodology must cleanly support heterogeneity. This paper will concentrate on code generation for DSP, but will describe a software architecture capable of adapting to such heterogeneous design problems.

A number of design styles can be used to develop signal processing software. One option, of course, is to rely on traditional high-level languages, notably C or Ada. Unfortunately, for many intensive signal processing applications, compilers for these languages are still unable to achieve the code efficiency demanded by designers. Twelve years after the appearance of programmable DSPs, most designers still prefer to program them in assembly language. The difficulty appears to be both in the languages themselves, which are not sufficiently specific to signal processing and poorly matched to fixed point data types; and in the processor architectures, which include features that compilers cannot easily support such as esoteric addressing modes (for example, bit reversed addressing for FFTs and hardware support for circular buffers). Numeric C [3] offers an interesting alternative by modifying the syntax of C to expose to the compiler much of the information it needs. Silage, an applicative language developed by Hilfinger at U. C. Berkeley, provides another alternative. The simple declarative semantics of the language and its fixed point data types make very efficient code generation possible [4]. The Mentor/EDC DSPStation uses Silage for its underlying semantics.

We are pursuing a third alternative, embodied previously in the Gabriel system [5], and more recently implemented in the Ptolemy system [6]. In this methodology, hand written assembly code segments define functional operators on data streams. Code generation consists of two phases, scheduling and synthesis. In the scheduling phase, the functional operators are possibly partitioned for parallel execution, and for each target processor, a sequence of operator invocation is determined. In the synthesis phase, the hand-written assembly code segments (or alternatively, higher-level language code segments or a mixture of both) are stitched together. This methodology has recently been commercialized in the Comdisco DPC system [1] and will be commercialized in the CADIS Descartes [7] systems. The techniques we describe here are complementary to those in DPC and Descartes, and could, in principle, be used in combination. In particular, we focus on management of data passed between functional blocks when synchronous dataflow (SDF) [8] and dynamic dataflow semantics are used. DPC, by contrast, does not use dataflow semantics.

## 1.1 Overview of Ptolemy

Ptolemy relies heavily on the methodology of object-oriented programming (OOP) to support heterogeneity. The basic unit of modularity in Ptolemy is the Block[1], illustrated in figure 1. A Block contains a module of code (the `go()` method) that is invoked at run-time, typically examining data present at its input Portholes and generating data on its output Portholes. Depending on the model of computation, however, the functionality of the `go()` method can be very different; it may spawn processes, for example, or synthesize assembly code for a target processor. In code generation applications, which are the concern of this paper, the `go()` method always synthesizes code in some target language. Its invocation is directed by a Scheduler (another modular object). A Scheduler determines the operational semantics of a network of

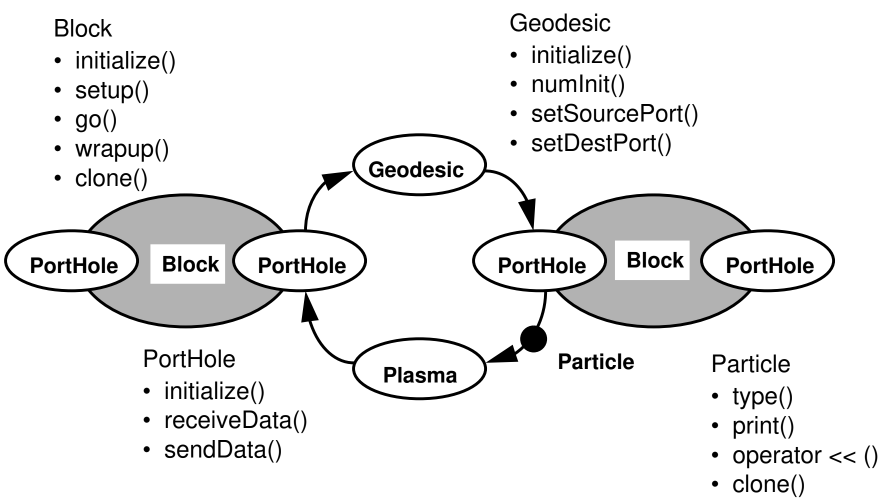


**Figure 1. Block objects in code generation applications of Ptolemy synthesize code in some target language. PortHoles and Geodesics provide methods for managing the exchange of data between blocks.**

1. When we capitalize a modular element, then it represents an object type. In object-oriented programming, objects encapsulate both data, the state of the object, and functions operating on that state, called methods.

Blocks. A third type of object, a Target, describes the specific features of a target for code generation. Blocks, Schedulers, and Targets can be designed by end users, lending generality while encouraging modularity. The hope is that Blocks will be well documented and stored in standard libraries; thus rendering them modular, reusable software components. The user-interface view of the system is an interconnected block diagram.

A conventional way to manage the complexity of a large system is to introduce a hierarchy in the description, as shown in figure 2. The lowest level (atomic) objects in Ptolemy are of type Star, derived from Block. A Star that generates code in some target language belongs to a *domain*, as explained below. The Stars in domain named "XXX" are of type XXXStar, derived from Star. A Galaxy, also derived from Block, contains other Blocks internally. A Galaxy may contain internally both Galaxies and Stars. A Galaxy may exist only as a descriptive tool, in that a Scheduler may ignore the hierarchy, viewing the entire network of blocks as flat. All our dataflow schedulers do this to maximize the visible concurrency. Alternatively, a Scheduler may make use of the hierarchy to minimize scheduling complexity or to structure synthesized code in a readable way. A third possibility we also exploit is for the scheduler to cluster the graph, creating a new hierarchy that reflects the natural looping structure of the code [9]. A Universe, which contains a

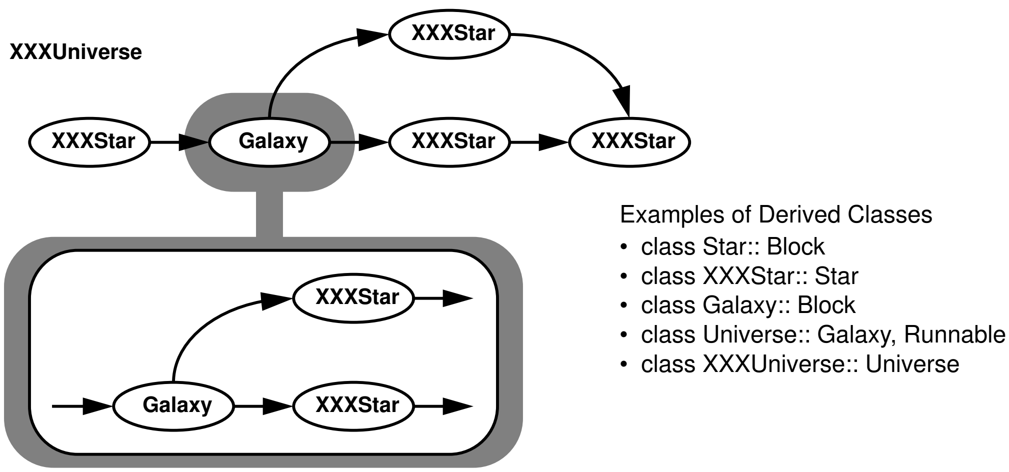


**Figure 2. A complete Ptolemy application (a Universe) consists of a network of Blocks. Blocks may be Stars (atomic) or Galaxies (composite). The "XXX" prefix symbolizes a particular domain (or model of computation).**

complete Ptolemy application, is a type of Galaxy. It is multiply derived from both Galaxy and class Runnable. The latter class contains methods for execution of simulation or synthesis of code.

In this paper, I will concentrate on one model of computation, synchronous dataflow. This is the model of computation for which we have best developed the code synthesis technology. I will first define these model of computation. Then I will introduce the modular element in Ptolemy, known as the domain, which encapsulates a single model of computation. Afterwards, I will introduce the code generation framework of Ptolemy, which allows definition of target architectures and the various interchangeable schedulers. After target architectures and domains are defined, I can then describe the atomic unit of an algorithm in Ptolemy, the Star, and the use of codeblocks (in the target language) for code generation. Next, the wormhole interface and how it relates to code generation will be described. I will then summarize the code generation procedure. Finally, I compare Ptolemy to other code generation environments.

Although this paper focuses on the current Ptolemy code generation domains, Ptolemy incorporates a rich set of simulation domains. Some of the domains currently defined are discrete event (DE), communication processes (CP), multi-threaded data flow (MTDF) and Thor (which will be described below). The Domain and the mechanism for co-existence of Domains are the primary abstractions that distinguish Ptolemy from otherwise comparable systems. For a description of the Ptolemy platform refer to [6].

### 1.1.1 DDF

Dynamic dataflow (DDF) is a data-driven model of computation originally proposed by Dennis [10]. Although frequently applied to the design of parallel architectures, it is also suitable as a programming model [11], and is particularly well-suited to signal processing applications with asynchronous operations. An equivalent model is embodied in the predecessor system Blosim [12, 13]. In DDF, Stars are enabled by data at their input PortHoles. That data may or may not be consumed by the Star when it fires, and the Star may or may not produce data on its outputs. More than one Star may be fired at one time if the Target supports this parallelism. We have used this domain to experiment with static scheduling of programs with run-time dynamics [14, 15].

### 1.1.2 SDF

Synchronous dataflow (SDF) [8] is a sub-Domain of DDF. SDF Stars consume and generate a static and known number of data tokens on each invocation. Since this is clearly a special case of DDF, any Star or Target that works under the SDF model will also work under the DDF model. However, an SDF Scheduler can take advantage of this static information to construct a schedule that can be used repeatedly. Such a Scheduler will not always work with DDF Stars. SDF is an appropriate model for multirate signal processing systems with rationally-related sampling rates throughout [15], and is the model used exclusively in Ptolemy's predecessor system Gabriel [5]. The advantages of SDF are ease of programming, since the availability of data tokens is static and does not need to be checked; a greater degree of setup-time syntax checking, since sample-rate inconsistencies are easily detected by the system; run-time efficiency, since the ordering of Block invocation is statically determined at setup-time rather dynamically at run-time; and automatic parallel scheduling [16-18].

## 1.2 Code Generation Domains

A Domain in Ptolemy consists of a set of Blocks and Targets, and associated Schedulers that conform to a common computational model. By "computational model" we mean the operational semantics governing how Blocks interact with one another. Furthermore, all Blocks and Targets of a code generation Domain target the same language; for example, Blocks that generate code for the Motorola 56000 using the SDF model of computation form their own domain[1]. A Scheduler will exploit knowledge of these semantics to order the execution of the

1. This definition of a Domain is different from the previous definition used in Ptolemy. When Ptolemy was solely a simulation environment, two distinct Domains would not share the same model of computation. Now, two distinct Domains can share the same model of computation as long as they target two distinct languages.

Blocks. SDF and DDF are domains related to one another as illustrated in figure 3. Stars and Targets are shown within each domain. The inner Domain (SDF) in figure 3 is an illustration of a sub-Domain, which implements a more specialized model of computation than the outer Domain (DDF). Hence all its Stars and Targets can also be used with the outer Domain. Schedulers can be associated with more than one Domain, but a Scheduler for a sub-Domain is not necessarily valid within the outer Domain.

For code generation, Domains are further subdivided according to the language synthesized. Hence, an SDF domain synthesizing C code is a domain that we call CGC (code generation in C). An SDF domain synthesizing assembly code for the Motorola DSP56000 family is called the CG56 domain. We have also developed SDF domains that synthesize assembly code for the Motorola DSP96000 family (CG96) and the Sproc multiprocessor DSP from Star Semiconductor. Finally, a Silage code generation domain is being used to couple to hardware synthesis tools developed at Berkeley [2].

As a simple example of how Blocks, Schedulers, and Targets can be mixed and matched, consider a set of Blocks that generate assembly language code for Motorola DSP56000 family processors. We might choose to use any of several Targets; examples of Targets that have been

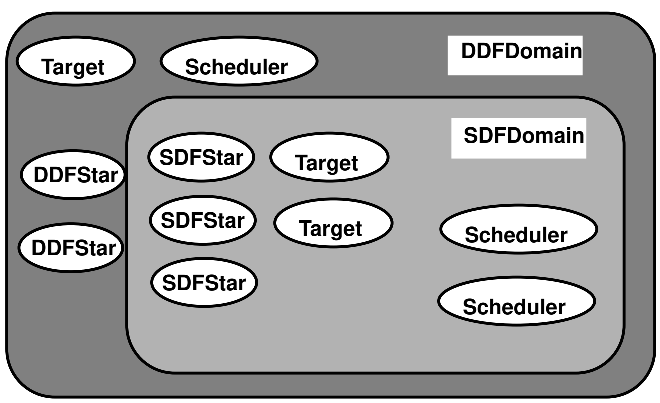


**Figure 3. A Domain (XXX) consists of a set of Stars, Targets and Schedulers that support a particular model of computation. A sub-Domain (YYY) may support a more specialized model of computation.**

implemented include one that runs the assembled code on a simulator on the workstation, one that describes an S-bus card with a single 56000 processor on a workstation, and one that describes a set of four interconnected processors on a single card. It is also possible to define targets that have not been built. In these cases the generated code runs on functional simulations of the processors in the Thor domain in Ptolemy [19]. Most targets have parameters that select what scheduler is to be used; we have several single- and multiple-processor Schedulers that use different algorithms for determining partitioning and order of execution of stars. These schedulers have no processor-specific information; they "ask" the Target to determine communication costs and "ask" the Block to determine execution time, resources needed, etc.

## 2.0 Code Generation with Ptolemy

### 2.1 General Framework

To use Ptolemy to implement an algorithm, the problem is represented as a hierarchical dataflow graph. Two interfaces are provided: a graphical interface based on VEM, the graphic editor that is part of U.C. Berkeley's Octtools CAD system [20], and a text interface based on Ousterhout's extensible interpreter language Tcl [21]. The user builds graphs hierarchically out of existing blocks, and may also link in user-written blocks by using Ptolemy's incremental linking facility. A special preprocessor makes user-written atomic blocks (stars) easier to produce.

While this paper focuses on code generation facilities, a key feature of Ptolemy is its ability to interface different models of computation. For example, code on a DSP board can interact with a discrete-event or logic simulation running on a workstation. Similarly, a register-transfer-level simulation of hardware (complete with programmable DSPs modeled functionally) can execute generated code and process signals synthesized in another Ptolemy domain. This gives Ptolemy most of its power when applied to hardware-software codesign. The interfacing mechanism that permits one model of computation, or domain, to interface cleanly with another is called a *wormhole*, after the theoretical cosmological phenomenon widely used in science fiction writing that may connect widely separated regions of space, or even different universes. This mechanism is described in [6, 19], and is explained in the context of code generation with a simple example given in section 2.5.

All code generation domains are derived from the CG domain. Only the derivative domains are of practical use for generating code. The stars in the CG domain itself can be thought of as “comment generators”; they are useful for testing and debugging schedulers and for little else. The CG domain is intended as a model and a collection of base classes for derivative domains. The code generation class hierarchy is designed to save work and to make the system more maintainable. Most of the work required to allocate memory for buffers, constants, tables, and to generate symbols that are required in code is completely processor-independent; thus these facilities are provided in generic classes.

In the following sections, I will introduce Targets and Stars and detail the methods and data structures needed to write new ones. I will first define a Ptolemy target, introducing the concepts of code streams, code generation methods, and wormhole methods. Next, I will detail what is required to write single-processor target. Afterwards I will define code generation stars and their respective methods. Following that I will describe the various methods which will generally use the `addCode()` method to piece together the codeblocks into the code streams. Finally I will document the various schedulers available in the code generation domains.

### 2.2 Targets

In Ptolemy, a Target class defines those features of an architecture pertinent to code generation. Each domain, which synthesizes a specific language such as C or Motorola 56000 assembly, has a simple target that will generate code and optionally compile or assemble the code. More elaborate Target definitions are derived from these. The more elaborate targets generate and run code on specific hardware platforms or on simulated hardware. Some examples that have been implemented are an S-56X[1] target and the CM5 from Thinking Machines. The latter is an example of a multiprocessor C language target. To define multiprocessor targets, the concept of Parent-Child target relationships is used. For example, the CM5 target contains an arbitrary

1. The S-56X is an S-bus card designed by Berkeley Camera Engineering and marketed by Ariel. It contains a Motorola DSP 56000 and a Xilinx FPGA.

number of C child targets. For our specific configuration of the CM5 at Berkeley, there are 128 child targets. In this paper we will focus on single-processor targets.

For any given code generation galaxy, a Target must be specified. The Target defines how the generated code will be collected, specifies and allocates resources such as memory, and defines code necessary for proper initialization of the platform. The Target will also specify how to compile and run the generated code. Optionally, it may also define wormholes (covered in section 2.5).

The derivation tree for all currently defined single-processor targets is shown in figure 8. At the top of the tree is the generic code generation target (CG). All code common to all code generation targets resides in the CG target. Methods defined here include virtual methods[1] to generate, display, compile and run the code, and a method to call these methods based on target or user specified parameters. The Assembly language target adds methods for the allocation of physical memory and interrupt handling. The higher level language target (HLL) contains methods to define and initialize variables, arrays, and include files.

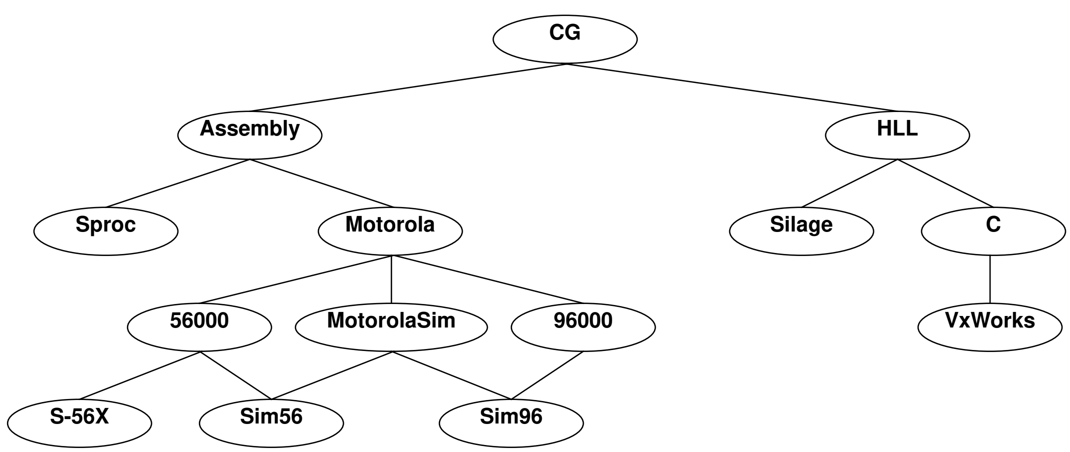


**Figure 4. Inheritance Tree for Single Processor Targets.**

1. In C++, a virtual method in a class is a method that can be optionally overloaded in derived classes in such a way that the appropriate function is selected at run time.

The object-oriented design of Ptolemy code generation makes target specification easy. For a typical target, the target writer must overload the `compileCode()` and `runCode()` methods. If the target is an assembly language target, the writer must also specify the memory. Multiple inheritance[1] can also be used to define similar targets. For example, as is shown in figure 1, both of the Motorola simulator targets are derived from a common Motorola simulator target for either the Sim56 or Sim96 target.

The base target for all code generation domains is the CGTarget, which represents a single processor by default. As the generic code generation target, the CGTarget class defines many common functions for code generation targets. Methods defined here include virtual methods to generate, display, compile, and run the code. Derived targets are free to redefine these virtual methods if necessary.

### 2.2.1 Code Streams

A code generation target manages code streams which are used to store star and target generated code. The CGTarget class has the two predefined code streams: `myCode` and `procedures`. Derived targets are free to add more code streams using the CGTarget method `addStream(stream-name)`. For example, the default CGC target defines six additional code streams.

The `addCode(code,stream name,<unique name>`) method of a CG star provides an interface to all the code streams (stream name and unique-name arguments are optional). This method defaults to adding code into the `myCode` stream. If a stream name is specified, `addCode()` looks up the stream using the `getStream(stream-name)` method and then adds the code into that stream. Furthermore, if a unique name is provided for the code, the code will only be added if no other code has previously been added with the given unique name. The method `addCode()` will return TRUE if the code-string has been added to the stream and otherwise will return FALSE.

Other methods, such as `addProcedure(code,<unique name>)` can be defined, to provide a more efficient or convenient interfaces to a specific code stream (in this case,

1. In C++, multiple inheritance means that a class has two or more parent classes.

procedures). With `addProcedure()` it becomes clear why unique names are necessary. Recall that `addProcedure()` is used to declare outside of the main body of the code. For example, say we wanted to write a function in C to multiply two numbers. The codeblock to do this could read:

```
codeblock(sillyMultiply){
   /* A silly function */
   double $sharedSymbol(silly,mult)(double a, double b){
      double m;
            m = a*b;
            return m;
      }
}
```

Note that in this codeblock we used the `$sharedSymbol` macro described in the section 2.3.1 on page 20. To add this code to the procedures stream, in the `initCode()` method of the star, we can call one of the following:

```
   addProcedure(sillyMultiply,"mult");
   addCode(sillyMultiply,"procedures","mult");
   getStream("procedures")->put(sillyMultiply,"mult");
```

As with `addCode()`, `addProcedure()` returns a TRUE or FALSE indicating whether the code was inserted into the code stream. Taking this into account, we could have added the code line by line:

```
if(addProcedure("/* A silly function */\n","mult")){
   addProcedure("double $sharedSymbol(silly,mult)(double a, double b)\n");
   addProcedure("{\n");
   addProcedure("\tdouble m;\n");
   addProcedure("\tm = a*b;\n");
   addProcedure("\treturn m;\n");
   addProcedure("}\n");
}
```

### 2.2.2 Target Code Generation Methods

Once the program graph is scheduled, the target generates the code in the virtual method `generateCode()`. (Note: code streams should be initialized before this method is called.) All the methods called by `generateCode()` are virtual, thus allowing for target customization. The `generateCode()` method then calls `allocateMemory()` which allocates the target resources. After resources are allocated, the `initCode()` method of the stars are called by `codeGenInit()`. The next step is to form the main loop by calling the method `mainLoopCode()`. The number of iteration cycles are determined by the argument of the "run" directive which a user specifies in pigi or in ptcl. To complete the body of the main loop, `go()` methods of stars are called in the scheduled order. After forming the main loop, the `wrapup()` methods of stars are called.

Now, all of the code has been generated; however, the code can be in multiple target streams. The `frameCode()` method is then called to piece the code streams and place its resultant into the `myCode` stream. Finally, the code is written to a file by the method `writeCode()`. The default file name is "code.output", and that file will be located in the directory specified by a target parameter, `destDirectory`.

Finally, since all of the code has been generated for a target, we are ready to compile, load, and execute the code. Derived targets should redefine the virtual methods `compileCode()`, `loadCode()`, and `runCode()` to do these operations. At times it does not make sense to have separate `loadCode()` and `runCode()` methods, and in these cases, these operations should be collapsed into the `runCode()` method.

### 2.2.3 Target Wormhole Methods

CGTarget defines virtual methods necessary to support wormholes have to support wormholes, a target should redefine the virtual methods, `sendWormData()`, `receiveWormData()`, `wormInputCode()`, and `wormOutputCode()`. The `sendWormData()` method sends data from the Ptolemy host to the target architecture. The `wormInputCode()` method is in charge of defining the code in the target language to read in the data from the Ptolemy host. The methods `receiveWormData()` and `wormOutputCode()` are similar except that they correspond to data moving in the opposite direction. Further wormhole discussion is deferred until section 2.5 on page 26.

## 2.3 Stars

Ptolemy has two basic types of stars: simulation stars and code generation stars. For purposes of this paper, discussion will be limited to code generation stars.

The derivation tree for all currently defined abstract star classes is shown in figure 5. By an abstract star class, we mean that the classes are never used to generate target language code directly. Instead, these classes define macro function expansion and functional interfaces to target specified code streams. The leaf nodes[1] of the tree are used as parents for user definable code generation stars. All methods that are common to all code generation stars reside in base code generation star class (CGStar). Similarly, all code common to assembly code generation stars is found in the assembly language star (AsmStar), and all code common to higher level languages is defined in HLLStar.

Of special interest is the class AnyAsmStar. Stars derived from AnyAsmStar can be utilized in any assembly code generation domain. These stars do not produce code; their purpose is to manipulate the input and/or output buffers connected to these stars. Currently, there are two AnyAsmStars: BlackHole and Fork. A BlackHole star is a data sink that discards its input data. Other code generation stars can check if any of their outputs are connected to a BlackHole, and then conditionally generate code based on this fact. Also, all input buffers to BlackHoles are mapped into one single memory location, so even if stars do not check to see if a BlackHole is connected to one of its outputs, minimal buffer memory is utilized. The other type of AnyAsmStar that exists is the Fork star. A Fork star splits the data path into two or more paths; however, all data paths can share a single buffer. A series of connected Fork stars with interspersed delays can be collapsed and maintained at the output buffer where the first Fork was connected. As can be seen, AnyAsmStars are defined where no target language specific code needs to be generated. Instead, wise buffer management can lead to a general solution applicable to all code generation domains.

For each of the leaf nodes in figure 5, there exist predefined star libraries. However, for most users' needs, these libraries will be insufficient. As a result, special attention has been given to make star writing in Ptolemy, like Gabriel, easy and systematic [22]. Unlike Gabriel and other code generators previously mentioned, Ptolemy is object oriented, thus allowing users to easily re-use code. For example, the C code generation domain has the family of stars fixed lattice filter, adaptive lattice filter, and a vocoder. Here the vocoder star was derived (in the sense of C++ derived classes) from the adaptive lattice filter, in turn derived from the fixed lattice. Karjalainen

1. For example, in figure 5, the leaf nodes are: Sproc, 56000, 96000, AnyAsm, Silage, and C.

in [23] states that object oriented programming environments are well suited for DSP programming methodology.

A typical user-defined code generation star will consist of portholes, states, codeblocks, a `setup()` method, an `initCode()` method, a `go()` method, a `wrapup()` method, and an `execTime()` method. Portholes, states and codeblocks are all data members of a star. Portholes specify the inputs and outputs of the star and their types. States define user settable parameters or internal memory states required in the generated code. Codeblocks are a pseudo code specification of the target language. By pseudo code, we mean that the codeblock is made up of the target language and star macro functions. These macro functions can be defined at any level of the inheritance tree. Macro functions include parameter value substitution, unique symbol generation with multiple scopes, and state reference substitution.

`Setup()`, `initCode()`, `go()`, `wrapup()`, and `execTime()` make up the virtual methods of a star. Users are free to write additional methods that are called from one of five methods listed. The differentiating trait between `setup()`, `initCode()`, `go()`, and `wrapup()` methods is when the method is called. The `setup()` method is called before the schedule is generated and before any memory is allocated. It is responsible for setting up information that will affect scheduling and memory allocation, such as the number of values that are read from a particular porthole or the size of an array state. The main use of the `setup()` method, as in SDF, is to tell the scheduler if more than one sample is to be accessed from a porthole with the `setSDFParams()` call. The `initCode()` method is called before the schedule is generated and after the memory is allocated; code generated by `initCode()` appears before the main loop.

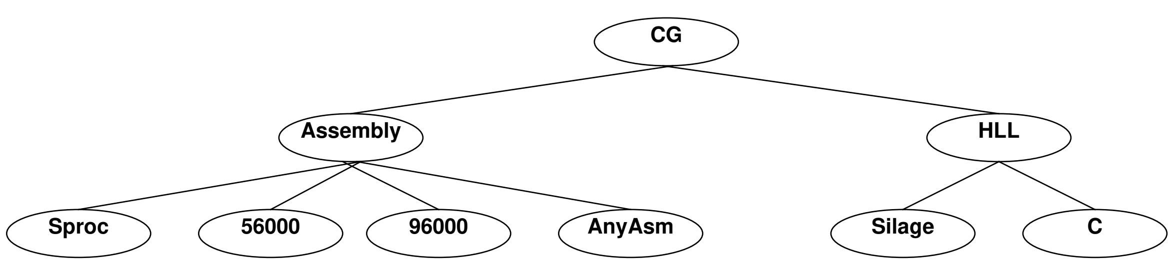


**Figure 5. Inheritance Tree for Code Generation Stars.**

The next method to be called is the `go()` method. This method is called directly from the scheduler. Hence the code generated in the `go()` method makes up the main loop code. Finally, the `wrapup()` method is called after the schedule has been completed, allowing the star to place code after the main loop code. For example, a typical use of this method in assembly code generation would be to define subroutines after the main loop code. The final virtual method that star writers may overload is `execTime()`. This method returns a number that indicates the approximate time to complete one firing of the star. This information is essential for the parallel schedulers.The better the `execTime()` estimates are for each star, the more efficient the parallel schedule becomes.

Stars are typically written not in C++ directly, but rather for a preprocessor called *ptlang*. This preprocessor generates the “standard boilerplate” necessary to properly initialize states and portholes, create codeblocks in a more natural manner, and to register the star with the system so that instances of it may be created by specifying the class name. It also generates documentation for the star.

### 2.3.1 Generic Code Generation Macros

In code generation stars, the inputs and outputs no longer hold values, but instead correspond to target resources where values will be stored (for example, memory locations/ registers in assembler generation, or global variables in c-code generation). A star writer can also define States which can specify the need for global resources.

A code generation star, however, does not have knowledge of the available global resources or the global variables/tables which have already been defined in the generated code. For star writers, a set of macros to access the global resources is provided. The macros are expanded in a language or target specific manner after the target has allocated the resources properly. In this section, we discuss the macros defined in the CGStar class.

`$ref(name)`: Returns a reference to a state or a port. If the argument, name, refers to a port, it is functionally equivalent to the “`name%0`” operator in the `SDF` simulation stars. If a star has a multi-porthole, say input, the first real porthole is input#1. To access the first porthole, we use

`$ref(input#1)` or `$ref(input#internal_state)` where internal_state is the name of a state that has the current value, 1.

`$ref(name,offset)`: Returns a reference to an array state or a port with an offset that is not negative. For a port, it is functionally equivalent to name%offset in SDF simulation stars.

`$val(state-name)`: Returns the current value of the state. If the state is an array state, the macro will return a string of all the elements of the array spaced by the new line character. The advantage of not using `$ref` macro in place of `$val` is that no additional target resources need to be allocated.

`$size(name)`: Returns the size of the state/port argument. The size of a non-array state is one; the size of a array state is the total number of elements in the array. The size of a port is the buffer size allocated to the port. The buffer size is usually larger than the number of tokens consumed or produced through that port.

`$starSymbol(name)`: Returns a unique label in the star instance scope. The instance scope is owned by a particular instance of that star in a graph. Furthermore, the scope is alive across all firings of that particular star. For example, two CG stars will have two distinct star instance scopes. As an example, we show some parts of ptlang file of the CGCPrinter star.

```
initCode{
   ...
   StringList s;
   s << "FILE* $starSymbol(fp);";
   addDeclaration(s);
   addInclude("<stdio.h>");
   addCode(openfile);
   ...
}

codeblock(openfile){
   if(!($starSymbol(fp)=fopen("$val(fileName)","w"))){
      fprintf(stderr,ERROR: cannot open output file for Printer star.\n");
   exit(1);
   }
}
```

The file pointer `fp` for a star instance should be unique globally, and the `$starSymbol` macro guarantees the uniqueness. Within the same star instance, the macro returns the same label.

`$sharedSymbol(list,name)`: Returns the symbol for name in the list scope. This macro is provided so that various stars in the graph can share the same data structures such as sin/cos lookup tables and conversion tables from linear to mu-law PCM encoder. These global data structures should be created and initialized once in the generated code. The macro `$sharedSymbol` does not provide the method to generate the code, but does provide the method to create a label for the code. To generate the code only once, refer to the discussion on code streams in section 2.2.1. An example where a shared symbol is used is in CGCPCM star is shown in figure 6.

The above code creates a conversion table and a conversion function from linear to mu-law PCM encoder. The conversion table is named `offset`, and belongs to the PCM class. The conversion function is named mulaw, and belongs to the same PCM class. Other stars can access that table or function by saying `$sharedSymbol(PCM,offset)` or `$sharedSymbol(PCM,mulaw)`. The `initCode()` method tries to put the sharedDeclarations codeblock into the global scope (by

```
codeblock (sharedDeclarations){
   int $sharedSymbol(PCM,offset)[8];
   /* Convert from linear to mu-law */
   int $sharedSymbol(PCM,mulaw)(x)
   double x;
   {
      double m;
      m = (pow(256.0,fabs(x)) - 1.0) / 255.0;
      return 4080.0 * m;
   }
}

codeblock (sharedInit){
   /* Initialize PCM offset table. */
   {
      int i;
      double x = 0.0;
      double dx = 0.125;
      for(i = 0; i < 8; i++, x += dx) {
         $sharedSymbol(PCM,offset)[i] = $sharedSymbol(PCM,mulaw)(x);
      }
   }
}

initCode {
   ...
   if (addGlobal(sharedDeclarations, "$sharedSymbol(PCM,PCM)"))
      addCode(sharedInit);
```

**Figure 6. Example of Shared Symbol Macro Usage**

`addGlobal()` method in the CGC domain). That codeblock is given a unique label by `$sharedSymbol(PCM,PCM)`. If the codeblock has not been previously defined, `addGlobal()` returns true, thus allowing `addCode(sharedInit)`. If there is more than one instance of the PCM star, only one instance will succeed in adding the code.

`$label(name), $codeblockSymbol(name)`: Returns a unique symbol in the codeblock scope. Both `$label` and `$codeblockSymbol` refer to the same macro expansion. The codeblock scope only lives as long as a codeblock is having code generated from it. Thus if a star uses `addCode()` more than once on a particular codeblock, all codeblock instances will have unique symbols. A example of where this is used in the CG56HostOut star.

```
codeblock(cbSingleBlocking) {
   $label(wait) jclr #m_htde,x:m_hsr,$label(wait)
   jclr #0,x:m_pbddr,$label(wait)
   movep $ref(input),x:m_htx
}

codeblock(cbMultiBlocking) {
   move #$addr(input),r0
   .LOOP #$val(samplesOutput)
   $label(wait) jclr #m_htde,x:m_hsr,$label(wait)
   jclr #0,x:m_pbddr,$label(wait)
   movep x:(r0)+,x:m_htx
   .ENDL
   nop
}
```

The above two codeblocks use a label named wait. The `$label` macro will assign unique strings for each codeblock.

To have “`$`” appear in the output code, put “`$$`” in the codeblock. For a domain where “`$`” is a frequently used character in the target language, it is possible to use a different character instead by redefining the virtual function `substChar()` (defined in CGStar) to return a different character.

It is also possible to introduce processor-specific macros, by overriding the virtual function `processMacro()` (rooted in CGStar) to process any macros it recognizes and defer substitution on the rest by calling its parent’s `processMacro()` method.

### 2.3.2 Assembly Code Generation Macros

Here we will present the additional and redefined macros available that have special meaning in assembly language code generation:

`$addr(name,<offset>)` This macro returns the numeric address in memory of the named object, without anything like (for the 56000) an "x:" or "y:" prefix. If the given quantity is allocated in a register (not yet supported) this function returns an error. It is also an error if the argument is undefined or is a state that is not assigned to memory (e.g. a parameter).

Note that this does not necessarily return the address of the beginning of a porthole buffer; it returns the "access point" to be used by this star invocation, and in cases where the star is fired multiple times, this will typically be different from execution to execution.

If the optional argument offset is specified, the macro returns an expression that references the location at the specified offset — wrapping around to the beginning of the buffer if that is necessary. Note that this wrapping works independently of whether the buffer is circularly aligned or not.

`$ref(name,<offset>)` This macro is much like `$addr(name)`, only the full expression used to refer to this object is returned, e.g. "`x:23`" for a 56000 if name is in x memory. If name is assigned to a register, this expression will return the corresponding register. The error conditions are the same as for `$addr`.

## 2.4 Schedulers

Given a Universe of functional blocks to be scheduled and a Target describing the topology and characteristics of the single- or multiple-processor system for which code is generated, it is the responsibility of the Scheduler object to perform some or all of the following functions:

- Determine which processor a given invocation of a given Block is executed on (for multiprocessor systems);
- Determine the order in which actors are to be executed on a processor;
- Arrange the execution of actors into standard control structures, like nested loops.

Not all schedulers perform all these functions (for example, we permit manual assignments of actors to processors if that is desired).

A key idea in Ptolemy is that there is no single scheduler that is expected to handle all situations. Users can write schedulers and can use them in conjunction with schedulers we have written. As with the rest of Ptolemy, schedulers are written following object-oriented design principals. Thus a user would never have to write a scheduler from ground up, and in fact the user is free to derive the new scheduler from even our most advanced schedulers. We have designed a suite of specialized schedulers that can be mixed and matched for specific applications. After the scheduling is performed, each processing element is assigned a set of blocks to be executed in a scheduler-determined order.

For targets consisting of a single processor, we provide two basic scheduling techniques. In the first approach, we simulate the execution of the graph on a dynamic dataflow scheduler and record the order in which the actors fire. To generate a periodic schedule, we first compute the number of firing of each actor in one *iteration* of the execution, which determines the number of appearances of the actor in the final scheduled list. An actor is called *runnable* when all input samples are available on its input arcs. If there is more than one actor runnable at the same time, the scheduler chooses one based on a certain criterion. The simplest strategy is to choose one randomly. There are many possible schedules for all but the most trivial graphs; the schedule chosen takes resource costs into account, such as the necessity of flushing registers and the amount of buffering required, into account (see [8] for detailed discussion of SDF scheduling). The Target then generates code by executing the actors in the sequence defined by this schedule. This is a quick and efficient approach unless there are large sample rate changes, in which case it corresponds to completely unrolling all loops. This scheduler is similar to one used in Gabriel [5].

The second approach we call “loop scheduling”. In this approach, actors that have the same sample rate are merged (wherever this will not cause deadlock) and loops are introduced to match the sample rates. The result is a hierarchical clustering; within each cluster, the techniques described above can be used to generate a schedule. The code then contains nested loop constructs together with sequences of code from the actors. The loop scheduling techniques used in Ptolemy are described in [9]; generalization of loop scheduling to include dynamic actors is discussed in [24].

### 2.5 Wormholes

A significant feature of Ptolemy is the capability of intermixing different domains or targets by wormholes. Suppose a code-generation domain lies in the SDF domain, where part of the application is to be run in simulation mode on the user’s workstation and the remainder of the application is to be downloaded to a DSP target system. When we schedule the actors that are to run in the outside SDF-simulation domain at compile-time, we generate, download, and run the code for the target architecture in the inside code-generation domain. For the purposes of this section, we will say “SDF domain” to refer to actors that are run in simulation mode, and “code generation domain” for actors for which code is generated.

In the example of figure 7-(a), a DSP target system is coded to estimate a power spectrum of a certain signal. At run-time, the estimated spectrum information is transferred to the host computer to be displayed on the screen. Thus, the host computer monitors the DSP system. In the next example in figure 7-(b), a DSP system performs a complicated filtering operation with a signal passed from the host computer, and sends the filtered result back to the host computer. In this case, the DSP hardware serves as a hardware accelerator for number crunching. By the wormhole mechanism in Ptolemy, as demonstrated in the above examples, we are able to make the host computer interact with the DSP system. In Ptolemy, a wormhole is an entity that, from the

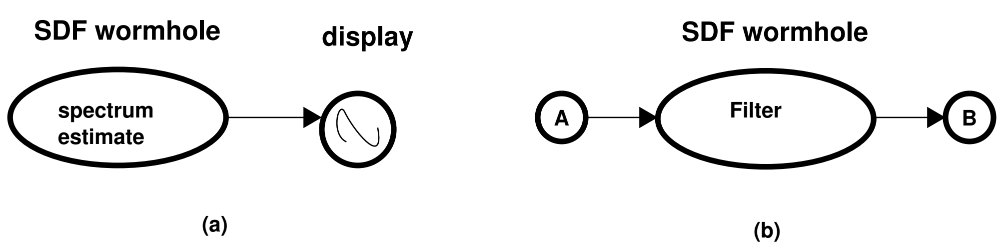


**Figure 7. Examples of Host-to-DSP interaction using wormholes.**

outside, obeys the semantics of one domain (in this case, it works like an SDF simulation actor), but on the inside, contains actors for another domain entirely.

Data communication between the host computer and the DSP target architecture is achieved in the wormhole boundary. In the SDF domain, data is transferred to the input porthole of the wormhole. The input porthole of a wormhole consists of two parts: one is visible from the outside SDF domain and the other is visible in the inside code-generation domain. The latter part of the porthole is designed in a target-specific manner, so that it sends the incoming data to the target architecture. In the output porthole of the wormhole, the inner part corresponding to the inside code-generation domain receives the data from the DSP hardware, which is transferred to the outer part visible from the outside SDF domain. In summary, for each target architecture, we can optionally design target specific wormholes to communicate data with the Ptolemy simulation environment; all that is needed to create this capability for a new Target is to write a pair of routines for transferring data that use a standard interface.

The interface code is generated by virtual target methods(`wormInputCode()`, `wormOutputCode()`), and the actual data transfer is also performed by other target methods (`sendWormData()`, and `receiveWormData()`). These methods were described in section 2.5. Unlike the simulation domains, the EventHorizon classes for the CG domain are not involved in the actual data communication, but perform other functions such as input data synchronization. A code-generation wormhole is only fired when all inputs are available from the simulation domain.

An example of a universe that contains a SDF wormhole interfacing to the DSP target is shown in figure 8. The SDF universe is used to here to display the output from a application running on the S-56X. The algorithm running on the DSP card is shown in figure 8. This is a very simple application, where a tone is being generated; the original signal, a upsampled (x 2) version and a downsampled (x 2) version is returned to the parent SDF universe. The code generated for the application shown in figure 8 is listed in section 7.1.

# 3.0 Summary of Code Generation Procedure

In this section we will review how the various modules of the Ptolemy platform interact to generate code for a target application. The code generation procedure is detailed in figure 10. First, the `setup()` method is called for all blocks relevant to particular application. This allows the schedulers, target modules, and stars to initialize internal variables. Next, the schedule pass is

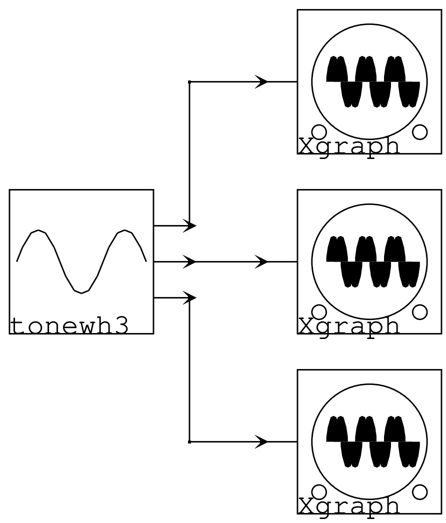


**Figure 8. SDF Universe containing a multirate S-56X Galaxy.**

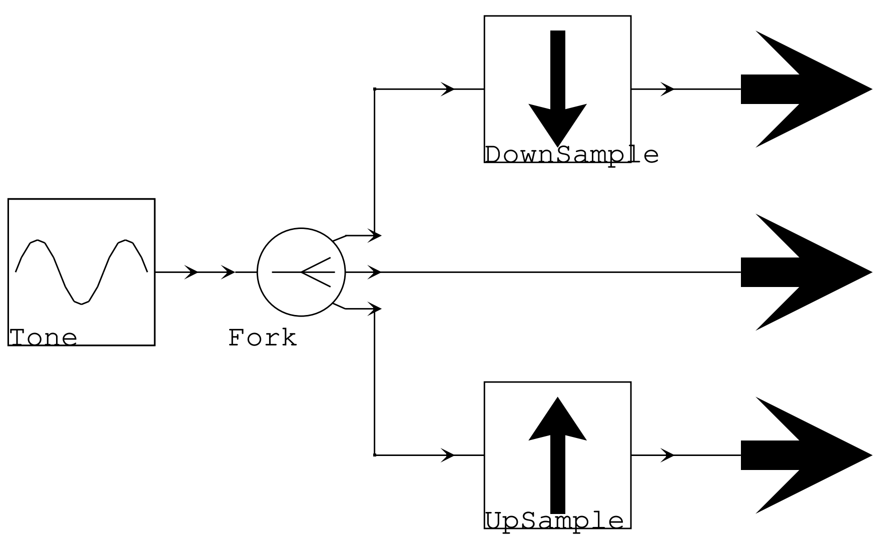


**Figure 9. Multirate S-56X Wormhole.**

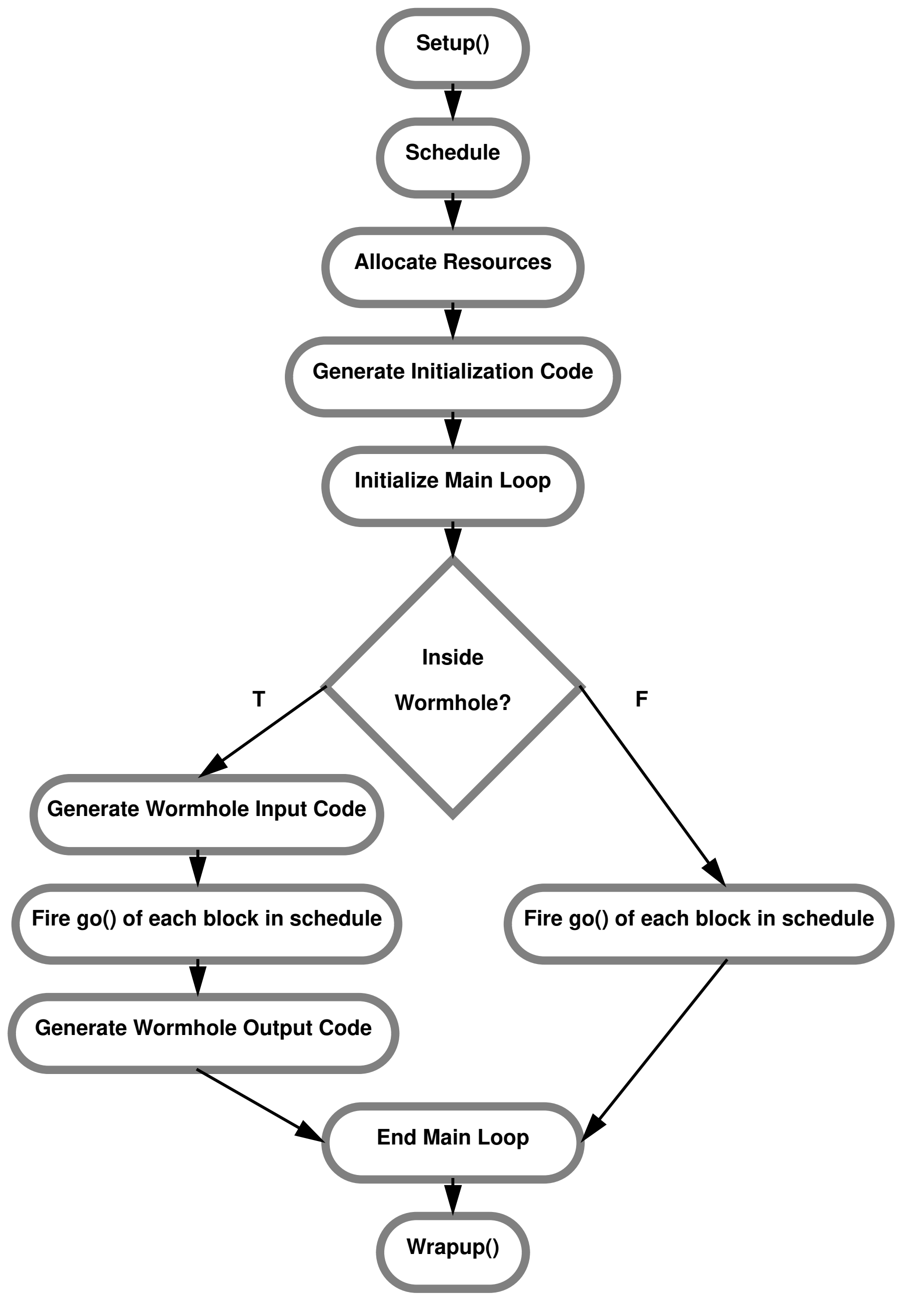


**Figure 10. Code Generation Procedure**

done. The scheduler returns a list that details the firing order of the blocks in a particular application. Based on this schedule, the resources can be allocated. In the case of assembly code, the memory is allocated as well. Note, the resource allocation stage must follow the scheduling stage so that the buffer lengths are known. Now we are ready to generate the initialization code for the given application. At this point, the `initCode()` method of all the blocks are fired. Finally, we are ready to generate the main loop code.

First we initialize the main loop. Notice that the code generation algorithm forks into two different paths, one signifying that the code currently being generated is intended for a target on the inside of a wormhole, and the other for applications not running inside a wormhole. If we are inside of a wormhole, we generate code to read the data from the Ptolemy Universal construct. Then we generate the main loop code and finally generate code to write the data into the Ptolemy construct. The wormhole code is written is a way which automatically synchronizes the DSP system and the host workstation. If we are not inside a wormhole, we simply generate the main loop code. Finally, we close the main loop and then fire the `wrapup()` methods of all of the blocks relevant to a particular application.

## 4.0 An Application: Adaptive PCM Coding

In this section, I will detail a simple application developed in the CG56 domain. An adaptive DPCM speech coder/decoder system was implemented using the S56XTarget. This target produces two files, one specifying the assembly code and the specifying the asychronous input output user interface. For a listing of each of these files see sections 7.2 and 7.3. The code generated runs in real-time on a Motorola 56000 DSP card installed in the SparcStation. This card is connected to an A/D and D/A that run at 8 kHz producing/consuming 16 bit samples. The coder allows various levels of quantization that can be readily interchanged at run time. The ADPCM coder is one implementation in the broad family of adaptive predictive coders (APC). First, a review of DPCM coders will be presented. For a more detailed derivation of ADPCM coders see [25].

A DPCM coder codes speech by quantizing a difference signal. Let $x(n)$ denote the original speech signal, then the difference signal is defined as $d(n) = x(n) - \hat{x}(n)$ where $\hat{x}(n)$ is the prediction of $x(n)$. Using a DPCM coder results in a coding gain known as the predictive gain, $G_P = \sigma_x^2 / \sigma_d^2$. A simplified DPCM coder is shown in figure 11. $H(e^{jw})$ is known as the prediction filter.

In order to be able to quantize $d(n)$, a feedback-around-quantizer structure (figure 12) is used so both of the prediction filters have access to the same information. This structure allows the use of an adaptive predictive filter where both filters adapt in unison, thus avoiding the need to transmit the predictor coefficients. In the system implemented, least mean square (LMS) adaptive filters are used for the prediction filters. This adaptive algorithm uses the instantaneous mean square error to adapt the filter coefficients.

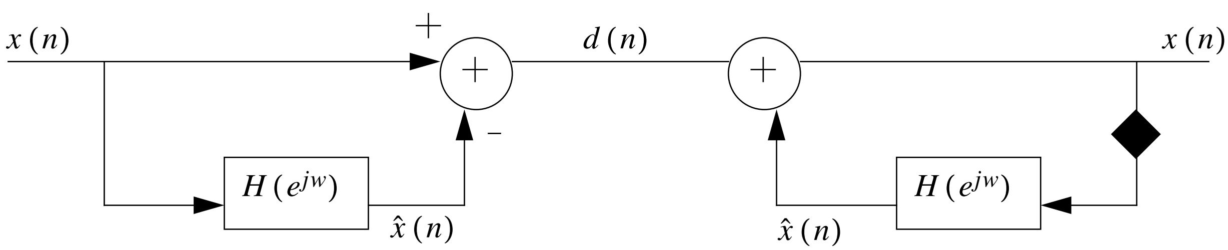


**Figure 11. A Simplified DPCM coder/decoder system.**

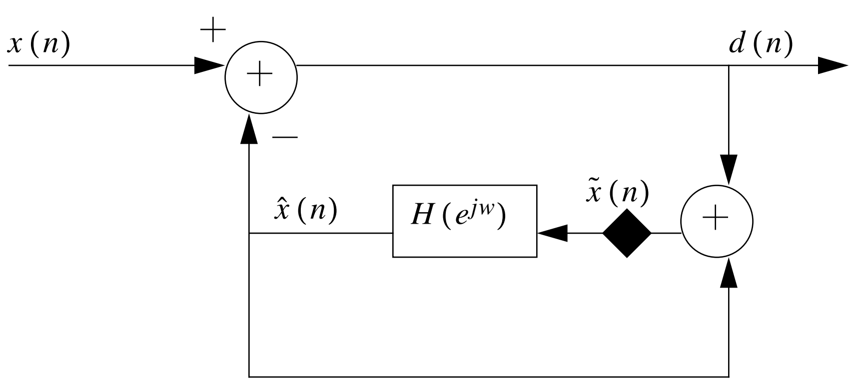


**Figure 12. A Feedback-around-quantizer coder.**

The coder and decoder for the system are shown in figures 13 and 14. The quantizer shown in figure 13 is a galaxy in Ptolemy. Inside this galaxy, there is a system of 4 quantizers feeding into a multiplexer. The multiplexer is controlled by the user interface shown in figure 15. Other parameters controllable here are an optional one second delay on the processed speech and a multiplicative constant applied to the quantizers to control their respective quantization and threshold levels. This constant allows the user to dynamically change the quantization parameters and instantly hear the results. A quantization range that is too large or too small impairs system performance. Thus with the slider, the user is able to fine tune the system.

The system produces intelligible speech at both 1 (8 kbs) and 2 (16 kbs) bit quantizations. At 3 bit quantization (24 kbs), the quality of the speech is very good. At 4 (32 kbs) bit quantization or no quantization, the speech quality is excellent.

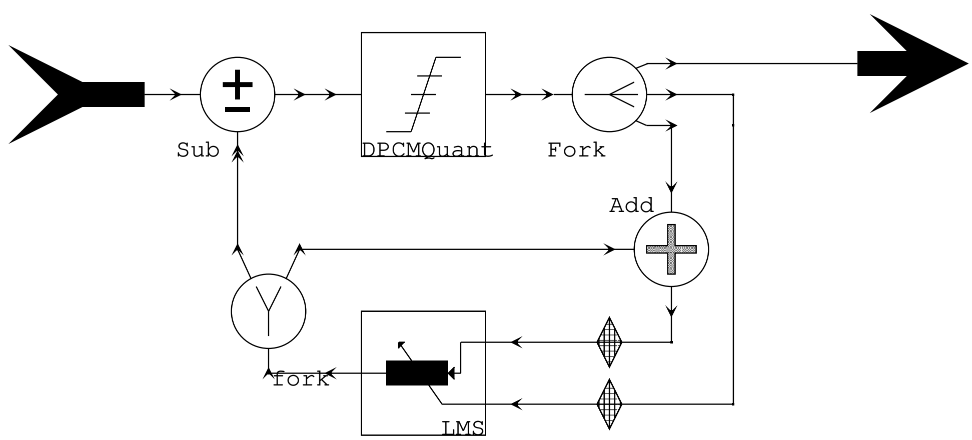


**Figure 13. ADPCM Coder**

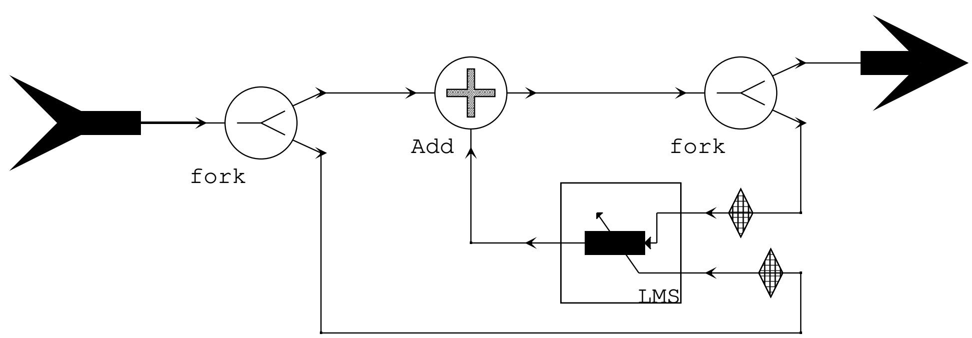


**Figure 14. ADPCM Decoder**

# 5.0 Conclusions

In this paper, we have introduced the code generation aspects of Ptolemy. It has been demonstrated that this platform provides an extensible signal processing code generation environment. Given the object-oriented design, Ptolemy allows the user to easily define new targets, stars, and schedulers. Once new blocks are defined they are easily incorporated into the Ptolemy environment, promoting code reuse. The *ptlang* preprocessor makes target and star writing systematic, especially for those unfamiliar with C++ or the Ptolemy kernel.

Comparing Ptolemy to the other DSP code generation platforms such Comdisco DPC [1], Mentor DSP Station, and Descartes [7], is difficult since we have addressed somewhat orthogonal issues. Some of these other code generators will do better in terms of efficiency for most SDF assembly language dataflow graphs. The reason for this lies in the fact that we have not implemented register allocation. We will be incorporating register allocation in the near future (see section 6.0). We can, however, compare Ptolemy to the other code generators in terms of features.

The major differences concern the handling of multirate signal processing. To implement a multirate graph, the Comdisco system uses "hold" signals on blocks. This introduces run-time conditional branching whenever the hold pins are connected. Unfortunately, the conditional branching is required even if the control flow is totally predictable at compile time. The Mentor DSP Station is built on top of the Silage language, which has only a limited mechanism for

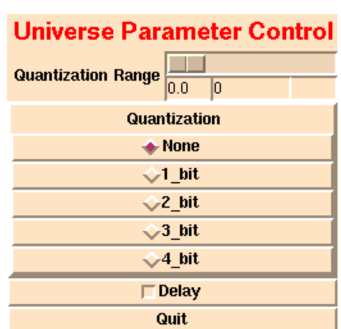


**Figure 15. Run Time User Interface**

expressing multirate systems. Silage contains upsample and downsample operators; however, it is impossible to write a polyphase multirate FIR filter block. To efficiently implement a multirate FIR filter (getting a polyphase implementation), Silage relies on dead-code elimination by the compiler. It is not clear how effective today's compilers would be in eliminating this dead-code. In Ptolemy, a polyphase FIR filter would simply be defined as a star, thus producing no dead-code.

Significant features distinguishing Ptolemy are the modularity gained from its object oriented design and its support for heterogenous architectures. We already support many scheduling algorithms. It is simple to test new scheduling heuristics and contrast those results with the supported schedulers. Also, we are not constrained to one particular scheduler for a signal processing application. Thus, a user is able to choose different schedulers for the various child-targets or domains in a single DSP application. For all other systems that we are aware of, a single scheduler is an integral part of the system.

The parallel schedulers are of particular interest. Here we are able to split, under special circumstances, the various invocations of a star instance over multiple processors. To do this we have defined Spread, Collect, Send and Receive stars. A great deal of support is provided for heterogeneous targets. For example, when a heterogeneous target specification is designed, previously defined targets can be used as the basic building blocks to more complex systems. The building block targets, in turn, can be either single-processor or multiple-processor targets.

## 6.0 Future Work

Although code generation is beginning to mature in Ptolemy, it is by no means finished. We have only begun to explore buffer management techniques to use memory more efficiently, Currently, in the assembly language domains, all stars must communicate through memory, not registers. Hence, the more fine-grained a star is, the more penalty it suffers. For example, a simple add star must first read in its two inputs from memory and then write its output to memory. Even though a simple operation like add might take one cycle on a DSP, the add could potentially take four or more cycles. Future versions of Ptolemy will use registers to exchange data, as done in [1].

Because there are no data-dependent decisions in the SDF domain, it is possible in principle to do more efficient register allocation than can be done for more conventional high-level languages (although since the problem of optimal register allocation, like so many others in this area, has combinatorial complexity, heuristics still must be used).

Work is in progress to extend our code generation techniques to support more general models of computation, such as the token flow model [24] and dynamic constructs [14]. We are also looking into developing tools to evaluate performance, and facilitate hand tuning of the generated code.

# 7.0 Appendix: Generated Code

## 7.1 S-56X Wormhole Generated Assembly Code

```
;User: pino
;Date: Mon Apr 19 11:13:06 1993
;Target: S-56XWH
;Universe: tonewh3
   org p:
   ori    #03,mr    ;disable interrupts
   include '/home/ohm1/users/messer/ptolemy/lib/cg56/intequlc.asm'
   include '/home/ohm1/users/messer/ptolemy/lib/cg56/ioequlc.asm'
   include '/home/ohm1/users/messer/ptolemy/lib/cg56/s56xwh.asm'
;initialization code from star wormHole3.tonewh31.tonewh3.Tone1 (class CG56Tone)
;initialization for state wormHole3.tonewh31.tonewh3.Tone1.state1
   org    y:1
   dc     0.0
;initialization for state wormHole3.tonewh31.tonewh3.Tone1.state2
   org    x:8
   dc     0.0626666167821521
   org p:
;initialization code from star wormHole3.tonewh31.tonewh3.UpSample1 (class CG56UpSample)
   move   #0,r1
   move   #0.0,a
   rep    #4
   move   a,x:(r1)+
;initialization code from star wormHole3.tonewh31.tonewh3.DownSample1 (class CG56DownSample)
;initialization code from star wormHole3.tonewh31.tonewh3.Fork.output=31 (class AnyAsmFork)
   andi   #$fc,mr   ;enable interrupts
LOOP_0
;code from star wormHole3.tonewh31.tonewh3.Tone1 (class CG56Tone)
   move   x:8,x1
   move   y:1,a
   move   #0.992114701314478,x0
   mac    -x1,x0,a  x1,x:4
   neg    a
   mac    x1,x0,a             x1,y:1
   move   a,x:8
;code from star wormHole3.tonewh31.tonewh3.Fork.output=31 (class AnyAsmFork)
;code from star wormHole3.tonewh31.tonewh3.UpSample1 (class CG56UpSample)
```

```
   move   x:4,x0
   move   x0,x:0
;code from star wormHole3.tonewh31.tonewh3.Tone1 (class CG56Tone)
   move   x:8,x1
   move   y:1,a
   move   #0.992114701314478,x0
   mac    -x1,x0,a  x1,x:5
   neg    a
   mac    x1,x0,a             x1,y:1
   move   a,x:8
;code from star wormHole3.tonewh31.tonewh3.Fork.output=31 (class AnyAsmFork)
;code from star wormHole3.tonewh31.tonewh3.UpSample1 (class CG56UpSample)
   move   x:5,x0
   move   x0,x:2
;code from star wormHole3.tonewh31.tonewh3.DownSample1 (class CG56DownSample)
   move   x:4,x0
   move   x0,x:9
; Output worm code for output#2
initial_wait_1
   move   y:WordCnt,a; get word count
   tst    a
   jeq    initial_wait_1
   jclr   #m_dma,x:m_hsr,initial_wait_1
   move #4,r0;read starting location address
   do     a,WHL_2
wait_3jclr#m_htde,x:m_hsr,wait_3;wait for host port available
   movep  x:(r0)+,x:m_htx
WHL_2nop
   move   #0,a
   move   a,y:WordCnt
   nop
; Output worm code for output
initial_wait_4
   move   y:WordCnt,a; get word count
   tst    a
   jeq    initial_wait_4
   jclr   #m_dma,x:m_hsr,initial_wait_4
wait_5jclr#m_htde,x:m_hsr,wait_5;wait for host port avail
   movep  x:9,x:m_htx
   move   #0,a
   move   a,y:WordCnt
   nop
; Output worm code for output
initial_wait_6
   move   y:WordCnt,a; get word count
   tst    a
   jeq    initial_wait_6
   jclr   #m_dma,x:m_hsr,initial_wait_6
   move #0,r0;read starting location address
   do     a,WHL_7
wait_8jclr#m_htde,x:m_hsr,wait_8;wait for host port available
   movep  x:(r0)+,x:m_htx
WHL_7nop
   move   #0,a
   move   a,y:WordCnt
   nop
   jmp    LOOP_0
   jmp    ERROR
; --------------------- Symmetric memory map:
; --------------------- x memory map:
; Loc 0, length 4, port wormHole3.tonewh31.tonewh3.UpSample1(output), type ANYTYPE
; Loc 4, length 2, port wormHole3.tonewh31.tonewh3.Fork.output=31(input), type ANYTYPE
; Loc 6, length 2, port wormHole3.tonewh31.tonewh3.Fork.output=31(output#2), type ANYTYPE
; Loc 8, length 1, state wormHole3.tonewh31.tonewh3.Tone1(state2), type FIX
```

```
; Loc 9, length 1, port wormHole3.tonewh31.tonewh3.DownSample1(output), type ANYTYPE
; --------------------- y memory map:
; Loc 1, length 1, state wormHole3.tonewh31.tonewh3.Tone1(state1), type FIX
```

## 7.2 ADPCM Generated Assembly Code

```
;User: pino
;Date: Wed Mar 17 17:19:46 1993
;Target: S-56X
;Universe: DPCM
   org p:
   ori    #03,mr    ;disable interrupts
   include '/home/ohm1/users/messer/ptolemy/lib/cg56/intequlc.asm'
   include '/home/ohm1/users/messer/ptolemy/lib/cg56/ioequlc.asm'
   include '/home/ohm1/users/messer/ptolemy/lib/cg56/s56x.asm'
;initialization code from star DPCM.monoADDA1.Fork.output=21 (class AnyAsmFork)
;initialization code from star DPCM.monoADDA1.SSI1 (class CG56SSI)
;initialization for state DPCM.monoADDA1.SSI1.missCnt
   org    y:15
   dc     0
   org p:
ssi_0_saveregequ68
ssi_0_buflenequ     8
ssi_0_bufferequ     0
ssi_0_recv_sptrequ74
ssi_0_recv_iptrequ76
ssi_0_xmit_sptrequ75
ssi_0_xmit_iptrequ77
ssi_0_dualbufequ0

; Initialize all the pointers to the right place.
; Note that recv&xmit bufs are at the same add but recv in x: and xmit in y:
 org x:74
 dc ssi_0_buffer
 org x:76
 dc ssi_0_buffer
 org x:75
 dc ssi_0_buffer
 org x:77
 dc ssi_0_buffer
 org p:

 move #ssi_0_buffer,r0
   .LOOP  #ssi_0_buflen
 bset #0,x:(r0); empty recv buf by setting bit 0
 bclr #0,y:(r0)+; fill xmit buf by clearing bit 0
   .ENDL
SAVEPC_3equ*
; SSI receive data interrupt vector
 org p:i_ssird
 jsr ssi_0_intr
   org    p:SAVEPC_3
SAVEPC_4equ*
; SSI receive data w/ exception interrupt vector
; XXX: this is wrong!!!
 org p:i_ssirde
 jsr ssi_0_intr
   org    p:SAVEPC_4
   movep  #16640,x:m_cra
   movep  #14848,x:m_crb
; Configure Port C pins 8-5 as SSI pins
 bset #8,x:m_pcc; STD
 bset #7,x:m_pcc; SRD
```

```
 bset #6,x:m_pcc; SCK (bit clock)
 bset #5,x:m_pcc; SC2 (frame clock)
; Configure Port C pins 2-0 as raw data pins (normally SCI)
 bclr #2,x:m_pcc
 bclr #1,x:m_pcc
 bclr #0,x:m_pcc
 bset #2,x:m_pcddr; as outputs
 bset #1,x:m_pcddr
 bset #0,x:m_pcddr
 bset #m_ssl0,x:m_ipr; set SSI IPL 2
 bset #m_ssl1,x:m_ipr
 bset #m_srie,x:m_crb ; enable SSI rx interupts
;initialization code from star DPCM.monoADDA1.BlackHole1 (class AnyAsmBlackHole)
;initialization code from star DPCM.DPCMTX1.DPCMQuant1.Fork.output=41 (class AnyAsmFork)
;initialization code from star DPCM.DPCMTX1.DPCMQuant1.QuantRange1 (class CG56QuantRange)
;initialization for state DPCM.DPCMTX1.DPCMQuant1.QuantRange1.thresholds
   org    x:81
   dc     0.0
;initialization for state DPCM.DPCMTX1.DPCMQuant1.QuantRange1.levels
   org    y:73
   dc     -0.5
   dc     0.5
   org p:
;initialization code from star DPCM.DPCMTX1.DPCMQuant1.QuantRange2 (class CG56QuantRange)
;initialization for state DPCM.DPCMTX1.DPCMQuant1.QuantRange2.thresholds
   org    x:71
   dc     -0.5
   dc     0.5
;initialization for state DPCM.DPCMTX1.DPCMQuant1.QuantRange2.levels
   org    y:70
   dc     -1.0
   dc     0.0
   dc     0.99999988079071
   org p:
;initialization code from star DPCM.DPCMTX1.DPCMQuant1.QuantRange3 (class CG56QuantRange)
;initialization for state DPCM.DPCMTX1.DPCMQuant1.QuantRange3.thresholds
   org    x:62
   dc     -0.7
   dc     -0.42
   dc     -0.14
   dc     0.14
   dc     0.42
   dc     0.7
;initialization for state DPCM.DPCMTX1.DPCMQuant1.QuantRange3.levels
   org    y:63
   dc     -0.84
   dc     -0.56
   dc     -0.28
   dc     0.0
   dc     0.28
   dc     0.56
   dc     0.84
   org p:
;initialization code from star DPCM.DPCMTX1.DPCMQuant1.HostSlider1 (class CG56HostSlider)
;initialization for state DPCM.DPCMTX1.DPCMQuant1.HostSlider1.value
   org    x:82
   dc     0.0
   org p:
;initialization  code  from  star  DPCM.DPCMTX1.DPCMQuant1.switch51.HostMButton1  (class
CG56HostMButton)
;initialization for state DPCM.DPCMTX1.DPCMQuant1.switch51.HostMButton1.value
   org    x:83
   dc     0.0
   org p:
```

```
;initializationcodefromstarDPCM.DPCMTX1.DPCMQuant1.switch51.Mux.input=51(classCG56Mux)
    org    x:8
    dc     80
    dc     85
    dc     86
    dc     87
    dc     88
    org    y:8
    dc     1-1
    dc     1-1
    dc     1-1
    dc     1-1
    dc     1-1
    org    p:
;initialization code from star DPCM.DPCMTX1.DPCMQuant1.QuantRange4 (class CG56QuantRange)
;initialization for state DPCM.DPCMTX1.DPCMQuant1.QuantRange4.thresholds
    org    x:48
    dc     -0.845
    dc     -0.715
    dc     -0.585
    dc     -0.455
    dc     -0.325
    dc     -0.195
    dc     -0.065
    dc     0.065
    dc     0.195
    dc     0.325
    dc     0.455
    dc     0.585
    dc     0.715
    dc     0.845
;initialization for state DPCM.DPCMTX1.DPCMQuant1.QuantRange4.levels
    org    y:48
    dc     -0.91
    dc     -0.78
    dc     -0.65
    dc     -0.52
    dc     -0.39
    dc     -0.26
    dc     -0.13
    dc     0.0
    dc     0.13
    dc     0.26
    dc     0.39
    dc     0.52
    dc     0.65
    dc     0.78
    dc     0.91
    org p:
;initialization code from star DPCM.DPCMTX1.DPCMQuant1.Fork.output=42 (class AnyAsmFork)
;initialization code from star DPCM.DPCMTX1.DPCMQuant1.auto-fork-60 (class AnyAsmFork)
;initialization code from star DPCM.DPCMTX1.Sub1 (class CG56Sub)
;initialization code from star DPCM.DPCMTX1.Add.input=21 (class CG56Add)
;initialization code from star DPCM.DPCMTX1.Fork.output=21 (class AnyAsmFork)
;initialization code from star DPCM.DPCMTX1.Fork.output=31 (class AnyAsmFork)
;initialization code from star DPCM.DPCMTX1.LMS1 (class CG56LMS)
;initialization for state DPCM.DPCMTX1.LMS1.coef
    org    x:16
    dc     0.99999988079071
    dc     0.0
    dc     0.0
    dc     0.0
    dc     0.0
    dc     0.0
```

```
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   org p:
; delayLine memory
 org y:16
 bsc 16,0
 org p:
; pointer to delay line into memory
 org y:75
 dc 16
 org p:
;initialization code from star DPCM.APCRx1.Add.input=22 (class CG56Add)
;initialization code from star DPCM.APCRx1.Fork.output=23 (class AnyAsmFork)
;initialization code from star DPCM.APCRx1.Fork.output=24 (class AnyAsmFork)
;initialization code from star DPCM.APCRx1.LMS2 (class CG56LMS)
;initialization for state DPCM.APCRx1.LMS2.coef
   org    x:32
   dc     0.99999988079071
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   dc     0.0
   org p:
; delayLine memory
 org y:32
 bsc 16,0
 org p:
; pointer to delay line into memory
 org y:76
 dc 32
 org p:
;initializationcodefromstarDPCM.SwitchDelay1.switch1.HostButton.buttonType=checkbutton1
(class CG56HostButton)
   org    x:96
   dc     0
   org    p:
;initialization code from star DPCM.SwitchDelay1.switch1.Mux.input=21 (class CG56Mux)
   org    x:13
   dc     95
   dc     98
   org    y:13
   dc     1-1
   dc     1-1
   org    p:
;initialization code from star DPCM.SwitchDelay1.Delay1 (class CG56Delay)
```

```
; initialize delay star
; pointer to internal buffer
   org    y:77
   dc     8192
   org    p:
;initialization code from star DPCM.SwitchDelay1.Fork.output=25 (class AnyAsmFork)
   andi   #$fc,mr   ;enable interrupts
LOOP_5
;code from star DPCM.DPCMTX1.LMS1 (class CG56LMS)
   ; initialize address registers for coef and delayLine
 move #16+16-1,r3
; insert here
 move y:75,r5 ; delayLine
 move #15,m5
 ; first adapt coefficients.
 ; multiply the error by the stepSize --> x0
 move #0.0001,x1
 move x:92,x0
 mpyr x0,x1,a
 move a,x0
 move x:(r3),b y:(r5)+,y0
 do #15,endloop_6
 macr x0,y0,b
 move b,x:(r3)-
 move x:(r3),b y:(r5)+,y0
endloop_6
 macr x0,y0,b
 move b,x:(r3)
; move current inputs into delayLine.
 move #93,r0
 move y:75,r5
 move x:(r0)+,y1
 move y1,y:(r5)+
; update delayLine pointer.
 move r5,y:75 ;oldest sample pointer
 ; now compute output.
 lua (r5)-,r5
 nop
 clr a x:(r3)+,x0 y:(r5)-,y0
 do #15,loop1_7
 mac x0,y0,a x:(r3)+,x0 y:(r5)-,y0
loop1_7
 macr x0,y0,a
 move a,x:91
 move m7,m5
;code  from  star  DPCM.SwitchDelay1.switch1.HostButton.buttonType=checkbutton1  (class
CG56HostButton)
   move   x:96,x0             ; move value to output
   move   x0,x:97
;code from star DPCM.APCRx1.LMS2 (class CG56LMS)
   ; initialize address registers for coef and delayLine
 move #32+16-1,r3
; insert here
 move y:76,r5 ; delayLine
 move #15,m5
 ; first adapt coefficients.
 ; multiply the error by the stepSize --> x0
 move #0.0001,x1
 move x:92,x0
 mpyr x0,x1,a
 move a,x0
 move x:(r3),b y:(r5)+,y0
 do #15,endloop_8
 macr x0,y0,b
```

```
 move b,x:(r3)-
 move x:(r3),b y:(r5)+,y0
endloop_8
 macr x0,y0,b
 move b,x:(r3)
; move current inputs into delayLine.
 move #95,r0
 move y:76,r5
 move x:(r0)+,y1
 move y1,y:(r5)+
; update delayLine pointer.
 move r5,y:76 ;oldest sample pointer
 ; now compute output.
 lua (r5)-,r5
 nop
 clr a x:(r3)+,x0 y:(r5)-,y0
 do #15,loop1_9
 mac x0,y0,a x:(r3)+,x0 y:(r5)-,y0
loop1_9
 macr x0,y0,a
 move a,x:94
 move m7,m5
;code from star DPCM.DPCMTX1.DPCMQuant1.switch51.HostMButton1 (class CG56HostMButton)
   move   x:83,x0             ; move value to output
   move   x0,x:84
;code from star DPCM.DPCMTX1.DPCMQuant1.HostSlider1 (class CG56HostSlider)
   move   x:82,x0             ; move value to output
   move   x0,x:89
;code from star DPCM.DPCMTX1.DPCMQuant1.Fork.output=42 (class AnyAsmFork)
;code from star DPCM.DPCMTX1.Fork.output=21 (class AnyAsmFork)
;code from star DPCM.DPCMTX1.Sub1 (class CG56Sub)
 move x:90,a
 move x:91,x0
 sub x0,a
 move a,x:80
;code from star DPCM.DPCMTX1.DPCMQuant1.Fork.output=41 (class AnyAsmFork)
;code from star DPCM.DPCMTX1.DPCMQuant1.auto-fork-60 (class AnyAsmFork)
;code from star DPCM.DPCMTX1.DPCMQuant1.QuantRange1 (class CG56QuantRange)
 move #<81,r0
 move #>73,r4
 move x:80,x0
   move   x:89,x1
 move x:(r0),y0
   move   y:(r4)+,y1
   mpy    x1,y0,a
   mpy    x1,y1,b
 cmpx0,a
 jgeterm_10
 move y:(r4),y1
   mpy    x1,y1,b
term_10
 move b,x:85
;code from star DPCM.DPCMTX1.DPCMQuant1.QuantRange2 (class CG56QuantRange)
 move #<71,r0
 move #>70,r4
 move x:80,x0
   move   x:89,x1
 move x:(r0)+,y0
   move   y:(r4)+,y1
 do #2-1,lab_11
   mpy    x1,y0,a
   mpy    x1,y1,b
 cmpx0,a
 jltagain_12
```

```
 enddo
 jmp term_13
again_12
 move x:(r0)+,y0
   move   y:(r4)+,y1
lab_11
 cmpx0,a
 jgeterm_13
 move y:(r4),y1
   mpy    x1,y1,b
term_13
 move b,x:86
;code from star DPCM.DPCMTX1.DPCMQuant1.QuantRange3 (class CG56QuantRange)
 move #<62,r0
 move #>63,r4
 move x:80,x0
   move   x:89,x1
 move x:(r0)+,y0
   move   y:(r4)+,y1
 do #6-1,lab_14
   mpy    x1,y0,a
   mpy    x1,y1,b
 cmpx0,a
 jltagain_15
 enddo
 jmp term_16
again_15
 move x:(r0)+,y0
   move   y:(r4)+,y1
lab_14
 cmpx0,a
 jgeterm_16
 move y:(r4),y1
   mpy    x1,y1,b
term_16
 move b,x:87
;code from star DPCM.DPCMTX1.DPCMQuant1.QuantRange4 (class CG56QuantRange)
 move #<48,r0
 move #>48,r4
 move x:80,x0
   move   x:89,x1
 move x:(r0)+,y0
   move   y:(r4)+,y1
 do #14-1,lab_17
   mpy    x1,y0,a
   mpy    x1,y1,b
 cmpx0,a
 jltagain_18
 enddo
 jmp term_19
again_18
 move x:(r0)+,y0
   move   y:(r4)+,y1
lab_17
 cmpx0,a
 jgeterm_19
 move y:(r4),y1
   mpy    x1,y1,b
term_19
 move b,x:88
;code from star DPCM.DPCMTX1.DPCMQuant1.switch51.Mux.input=51 (class CG56Mux)
   move   #8,r0
   move   x:84,n0
   nop
```

```
   move   x:(r0+n0),r2
   nop
   move   x:(r2),x0
   move   x0,x:92
;code from star DPCM.DPCMTX1.Fork.output=31 (class AnyAsmFork)
;code from star DPCM.APCRx1.Fork.output=24 (class AnyAsmFork)
;code from star DPCM.DPCMTX1.Add.input=21 (class CG56Add)
   move   x:92,x0   ; 1st input -> x0
   move   x:91,a              ; 2nd input -> a
   add    x0,a
   move   a,x:93              ; this move saturates
;code from star DPCM.APCRx1.Add.input=22 (class CG56Add)
   move   x:92,x0   ; 1st input -> x0
   move   x:94,a              ; 2nd input -> a
   add    x0,a
   move   a,x:95              ; this move saturates
;code from star DPCM.APCRx1.Fork.output=23 (class AnyAsmFork)
;code from star DPCM.SwitchDelay1.Fork.output=25 (class AnyAsmFork)
;code from star DPCM.SwitchDelay1.Delay1 (class CG56Delay)
 move x:95,x1
 move y:77,r0
 move #8000-1,m0
 move y:(r0),y0
 move x1,y:(r0)+
 move r0,y:77
 move y0,x:98
 move #-1,m0
;code from star DPCM.SwitchDelay1.switch1.Mux.input=21 (class CG56Mux)
   move   #13,r0
   move   x:97,n0
   nop
   move   x:(r0+n0),r2
   nop
   move   x:(r2),x0
   move   x0,x:73
;code from star DPCM.monoADDA1.Fork.output=21 (class AnyAsmFork)
;code from star DPCM.monoADDA1.SSI1 (class CG56SSI)
   move #ssi_0_buflen-1,m0
   move x:ssi_0_recv_sptr,r0
   nop
 jset #0,x:(r0),*; Wait for slot to have data
 move x:(r0),y0 ; Get sample from buffer
   IF     0
 bset #0,x:(r0)+ ; Mark slot as empty
   ENDIF
 move y0,x:90
   IF     0
    move   y0,x:78
   ENDIF
   move x:73,y0
   IF 0
 jclr #0,y:(r0),*; Wait for slot to be empty
   ENDIF
 move y0,y:(r0) ; Put data there
   IF 0
 bclr #0,y:(r0)+ ; Mark slot as full
   ELSE
 bset #0,x:(r0)+ ; Mark slot as empty
   ENDIF
 jset #0,x:(r0),*; Wait for slot to have data
 move x:(r0),y0 ; Get sample from buffer
   IF     0
 bset #0,x:(r0)+ ; Mark slot as empty
   ENDIF
```

```
 move y0,x:15
    IF     0
     move   y0,x:79
    ENDIF
    move x:73,y0
    IF 0
 jclr #0,y:(r0),*; Wait for slot to be empty
    ENDIF
 move y0,y:(r0) ; Put data there
    IF 0
 bclr #0,y:(r0)+ ; Mark slot as full
    ELSE
 bset #0,x:(r0)+ ; Mark slot as empty
    ENDIF
    move r0,x:ssi_0_recv_sptr
    move m7,m0
;code from star DPCM.monoADDA1.BlackHole1 (class AnyAsmBlackHole)
    jmp    LOOP_5
    jmp    ERROR
;Procedures Begin
; Interrupt handler for DPCM.monoADDA1.SSI1
ssi_0_intr
 move y0,x:ssi_0_savereg+0 ; Save y0, r0, m0
 move r0,x:ssi_0_savereg+1
 move m0,x:ssi_0_savereg+2
 move #ssi_0_buflen-1,m0
 move x:ssi_0_recv_iptr,r0; recv pointer
 move x:m_rx,y0
 jset #0,x:(r0),doRecv_1; make sure recv slot empty
    IF     1
 move#$123064,y0              ; its full...abort
 jmpERROR
    ELSE
     ; just drop recv sample in y0
     move  y:-(r0),y0          ; go back two (stereo): prev tx sample
     move  y:-(r0),y0
     move  y:15,r0
     move  y0,x:m_tx
     move  (r0)+
     move  r0,y:15
     jmp   done_2
    ENDIF
doRecv_1
 move y0,x:(r0)
 move y:(r0),y0
 bclr #0,x:(r0)+    ; mark slot as used
 move y0,x:m_tx
 move r0,x:ssi_0_recv_iptr; save updated pointer
done_2
 move x:ssi_0_savereg+0,y0 ; Restore y0, r0, m0
 move x:ssi_0_savereg+1,r0
 move x:ssi_0_savereg+2,m0
    rti
;Procedures End
; --------------------- Symmetric memory map:
; Loc 0, length 8, state DPCM.monoADDA1.SSI1(buffer), type FIXARRAY (circular)
; Loc 8, length 5, state DPCM.DPCMTX1.DPCMQuant1.switch51.Mux.input=51(ptrvec), type INTARRAY
; Loc 13, length 2, state DPCM.SwitchDelay1.switch1.Mux.input=21(ptrvec), type INTARRAY
; --------------------- x memory map:
; Loc 15, length 1, port DPCM.monoADDA1.BlackHole1(input), type ANYTYPE (circular)
; Loc 16, length 16, state DPCM.DPCMTX1.LMS1(coef), type FIXARRAY
; Loc 32, length 16, state DPCM.APCRx1.LMS2(coef), type FIXARRAY
; Loc 48, length 14, state DPCM.DPCMTX1.DPCMQuant1.QuantRange4(thresholds), type FIXARRAY
; Loc 62, length 6, state DPCM.DPCMTX1.DPCMQuant1.QuantRange3(thresholds), type FIXARRAY
```

```
; Loc 68, length 3, state DPCM.monoADDA1.SSI1(saveReg), type FIXARRAY
; Loc 71, length 2, state DPCM.DPCMTX1.DPCMQuant1.QuantRange2(thresholds), type FIXARRAY
; Loc 73, length 1, port DPCM.monoADDA1.Fork.output=21(input), type ANYTYPE
; Loc 74, length 1, state DPCM.monoADDA1.SSI1(recvStarPtr), type INT
; Loc 75, length 1, state DPCM.monoADDA1.SSI1(xmitStarPtr), type INT
; Loc 76, length 1, state DPCM.monoADDA1.SSI1(recvIntrPtr), type INT
; Loc 77, length 1, state DPCM.monoADDA1.SSI1(xmitIntrPtr), type INT
; Loc 78, length 1, state DPCM.monoADDA1.SSI1(prevOut1), type FIX
; Loc 79, length 1, state DPCM.monoADDA1.SSI1(prevOut2), type FIX
; Loc 80, length 1, port DPCM.DPCMTX1.DPCMQuant1.Fork.output=41(input), type ANYTYPE
; Loc 81, length 1, state DPCM.DPCMTX1.DPCMQuant1.QuantRange1(thresholds), type FIXARRAY
; Loc 82, length 1, state DPCM.DPCMTX1.DPCMQuant1.HostSlider1(value), type FIX
; Loc 83, length 1, state DPCM.DPCMTX1.DPCMQuant1.switch51.HostMButton1(value), type FIX
; Loc 84, length 1, port DPCM.DPCMTX1.DPCMQuant1.switch51.Mux.input=51(control), type INT
; Loc 85, length 1, port DPCM.DPCMTX1.DPCMQuant1.switch51.Mux.input=51(input#2), type ANYTYPE
; Loc 86, length 1, port DPCM.DPCMTX1.DPCMQuant1.switch51.Mux.input=51(input#3), type ANYTYPE
; Loc 87, length 1, port DPCM.DPCMTX1.DPCMQuant1.switch51.Mux.input=51(input#4), type ANYTYPE
; Loc 88, length 1, port DPCM.DPCMTX1.DPCMQuant1.switch51.Mux.input=51(input#5), type ANYTYPE
; Loc 89, length 1, port DPCM.DPCMTX1.DPCMQuant1.Fork.output=42(input), type ANYTYPE
; Loc 90, length 1, port DPCM.DPCMTX1.Sub1(pos), type FIX
; Loc 91, length 1, port DPCM.DPCMTX1.Fork.output=21(input), type ANYTYPE
; Loc 92, length 1, port DPCM.DPCMTX1.Fork.output=31(input), type ANYTYPE
; Loc 93, length 1, port DPCM.DPCMTX1.LMS1(input), type FIX
; Loc 94, length 1, port DPCM.APCRx1.Add.input=22(input#2), type FIX
; Loc 95, length 1, port DPCM.APCRx1.Fork.output=23(input), type ANYTYPE
; Loc 96, length 1, state DPCM.SwitchDelay1.switch1.HostButton.buttonType=checkbutton1(value),
type FIX
; Loc 97, length 1, port DPCM.SwitchDelay1.switch1.Mux.input=21(control), type INT
; Loc 98, length 1, port DPCM.SwitchDelay1.switch1.Mux.input=21(input#2), type ANYTYPE
; --------------------- y memory map:
; Loc 15, length 1, state DPCM.monoADDA1.SSI1(missCnt), type INT
; Loc 16, length 16, state DPCM.DPCMTX1.LMS1(delayLine), type INTARRAY (circular)
; Loc 32, length 16, state DPCM.APCRx1.LMS2(delayLine), type INTARRAY (circular)
; Loc 48, length 15, state DPCM.DPCMTX1.DPCMQuant1.QuantRange4(levels), type FIXARRAY
; Loc 63, length 7, state DPCM.DPCMTX1.DPCMQuant1.QuantRange3(levels), type FIXARRAY
; Loc 70, length 3, state DPCM.DPCMTX1.DPCMQuant1.QuantRange2(levels), type FIXARRAY
; Loc 73, length 2, state DPCM.DPCMTX1.DPCMQuant1.QuantRange1(levels), type FIXARRAY
; Loc 75, length 1, state DPCM.DPCMTX1.LMS1(delayLineStart), type INT
; Loc 76, length 1, state DPCM.APCRx1.LMS2(delayLineStart), type INT
; Loc 77, length 1, state DPCM.SwitchDelay1.Delay1(delayBufStart), type INT
; Loc 8192, length 8000, state DPCM.SwitchDelay1.Delay1(delayBuf), type FIXARRAY (circular)
```

### 7.3 ADPCM Generated Asychronous Input/Output (AIO) Code

```
aio_slider x:82 DPCM.DPCMTX1.DPCMQuant1.HostSlider1 "Quantization Range" 0.0 1.0 0.0 0.0 1.0
"linear"
aio_multibutton x:84 DPCM.DPCMTX1.DPCMQuant1.switch51.HostMButton1 {Quantization} {"None 0"
"1_bit 1" "2_bit 2" "3_bit 3" "4_bit 4"}
aio_checkbutton x:92 DPCM.SwitchDelay1.switch1.HostButton.buttonType=checkbutton1 {Delay} 0 1
0
aio_slider x:97 DPCM.adjustableGain1.HostSlider2 "Volume" 0.0 1.0 0.99999988079071 0.0
0.99999988079071 "linear"
aio_checkbutton x:105 DPCM.switch2.HostButton.buttonType=checkbutton1 {ADPCM} 0 1 0
```

# 8.0 References


[1] D.G. Powell, E. A.Lee, and W.C. Newman, "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," *International Conference on Acoustics, Speech and Signal Processing*, vol. 5, San Francisco, IEEE, 1992, p. 553-556.

[2] J.M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of datapath-intensive architectures," *IEEE Design & Test of Computers*, vol. 8, no. 2, 1991, p. 40-51.

[3] K.W. Leary and W. Waddington, "DSP/C: A Standard High Level Language for DSP and Numeric Processing," *International Conference on Acoustics, Speech and Signal Processing*, vol. 2, 1990, p. 1065-1068.

[4] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, and H. De Man, "DSP specification using the Silage language," *International Conference on Acoustics, Speech and Signal Processing*, vol. 2, 1990, p. 1056-1060.

[5] J.C. Bier, E.E. Goei, W.H. Ho, P.D. Lapsley, M.P. O'Reilly, G.C. Sih, and E.A. Lee, "Gabriel: A design environment for DSP," *IEEE Micro*, vol. 10, no. 5, 1990, p. 28-45.

[6] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A Platform for Heterogeneous Simulation and Prototyping," *European Simulation Conference*, Copenhagen, Denmark, 1991.

[7] S. Ritz, M. Pankert, and H. Meyr, "High Level Software Sythesis for Signal Processing Systems," *International Conference on Application Specific Array Processors*, IEEE Computer Society Press, 1992, p. 679-693.

[8] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, 1987, p. 1235-1245.

[9] S.S. Bhattacharyya, "Scheduling synchronous dataflow graphs for efficient looping," *to appear in Journal of VLSI Signal Processing*, 1993.

[10] J.B. Dennis, "Data Flow Supercomputers," *IEEE Computer*, vol. 13, no. 11, 1980.

[11] A.L. Davis and R.M. Keller, "Data Flow Program Graphs," *IEEE Computer*, vol. 15, no. 2, 1982.

[12] D.G. Messerschmitt, "Structured Interconnection of Signal Processing Programs," *Globecom*, Atlanta, Georgia, 1984.

[13] D.G. Messerschmitt, "A Tool for Structured Functional Simulation," *IEEE Journal on Selected Areas in Communications*, vol. SAC-2, no. 1, 1984.

[14] S. Ha, *Compile-time scheduling of dataflow program graphs with dynamic constructs*, Ph.D. Dissertation, U.C. Berkeley, 1992.

[15] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Multirate signal processing in Ptolemy," *International Conference on Acoustics, Speech and Signal Processing*, vol. 2, New York, NY, USA, IEEE, 1991, p. 1245-1248.

[16] E.A. Lee and J.C. Bier, "Architectures for statically scheduled dataflow," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, 1990, p. 333-348.

[17] G.C. Sih and E.A. Lee, "Dynamic-level scheduling for heterogeneous processor networks," *Second IEEE Symposium on Parallel and Distributed Processing*, 1990, p. 42-49.

[18] G.C. Sih and E.A. Lee, "Declustering: A New Multiprocessor Scheduling Technique," *IEEE Transactions on Parallel and Distributed Systems*, 1992.

[19] A. Kalavade, "Hardware/Software Codesign using Ptolemy — A Case Study," *International Workshop on Hardware/Software Codesign*, Grassau, Germany, 1992.

[20] D.S. Harrison, P. Moore, R. Spickelmier, and A.R. Newton, "Data Management and Graphics Editing in the Berkeley Design Environment," *IEEE Internation Conference on Computer-Aided Design*, 1986.

[21] J.K. Ousterhout, "Tcl: An Embeddable Command Language," *Winter USENIX Conference*, 1990, p. 133-146.

[22] J.C. Bier and E.A. Lee, "Frigg: A Simulation Environment For Multiple-Processor DSP System Development," *International Conference on Computer Design: VLSI in Computers and Processors*, Washington, DC, USA, IEEE Computer Society Press, 1989, p. 280-283.

[23] M. Karjalainen, "DSP software integration by object-oriented programming: a case study of QuickSig," *IEEE ASSP Magazine*, vol. 7, no. 2, 1990, p. 21-31.

[24] J. Buck and E.A. Lee, "The Token Flow Model," *Data Flow Workshop*, Hamilton Island, Australia, 1992.

[25] N.S. Jayant and P. Noll, *Digital Coding of Waveforms*, New Jersey: Prentice-Hall, 1984.