May 6, 1993

Software Synthesis for DSP Using Ptolemy

José Luis Pino
Soonhoi Ha
Edward A. Lee
Joseph T. Buck

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, and generation of real-time prototypes. In this paper we will introduce the software synthesis aspects of the Ptolemy system. The environment presented here is both modular and extensible. Ptolemy allows the user to choose among various single- or multiple-processor schedulers.

Keywords

- DSP
- Heterogeneous system design
- code generation
- block-diagram environments
- system-level design

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 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 makes very efficient code generation realistically 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 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

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.

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 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, we will concentrate on only two models of computation, dynamic and synchronous dataflow. These are the models of computation for which we have best developed the code synthesis technology. We will first define these models of computation. Then we will introduce the modular element in Ptolemy, known as the domain, which encapsulates a single model of computation. Afterwards, we will introduce the code generation framework of Ptolemy, which allows definition of target architectures (both single and multiple-processor) and the various interchangeable schedulers. After a target architecture and domain are defined, we can then describe the atomic unit of an algorithm in Ptolemy, the Star, and the use of code blocks (in the target language) for code generation. Next, the various abstractions of interprocessor communication available in Ptolemy will be described: send/receive, spread/collect, and the wormhole interface. We will then summarize the code generation procedure. Finally, we will 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 design parallel architectures, it is also suitable as a programming model [11], and is particularly well-suited to signal processing that includes 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].

1.1.3 The token flow model (BDF)

We are also exploring a third possibility, called the token flow model or boolean-controlled dataflow, which extends the SDF model to permit data movement to depend on the values of certain Boolean tokens in the system. The intent is to preserve the compile-time scheduling properties of SDF but permit data-dependent execution. This work is very new (see [19]) and will not be discussed further in this paper.

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 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 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 [20]. 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-

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.

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 [21], and a text interface based on Ousterhout’s extensible interpreter language Tcl [22]. 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, 20], and is explained in the context of code generation in section 2.5.3.

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 number of C child targets. For our specific configuration of the CM5 at Berkeley, there are 128 child targets.

2.2.1 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 2.5.3).

The derivation tree for all currently defined single-processor targets is shown in figure 4. 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[2] 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.

The object-oriented design of Ptolemy code generation makes target specification easy. For a typical target, the target writer must overload the compile and run methods. If the target is an assembly language target, the writer must also specify the memory. Multiple inheritance[3] 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.

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.

2. 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.

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

2.2.2 Multiple-Processor Targets

Targets representing multiple processors are also derived from the CG target class. The base class for all homogeneous multiple-processor targets is called MultiTarget; a MultiTarget has a sequence of child Target objects to represent each of the individual processors. The decomposition of function is done in such a way that classes derived from MultiTarget represent the topology of the multi-processor network (communication costs between processors, schedules for use of communication facilities, etc.), and single-processor child targets can represent arbitrary types of processors. The resource allocation problem is divided between the parent target, representing the shared resources, and the child targets, representing the resources that are local to each processor.

We have implemented, or are in the process of implementing, both “abstract” and “concrete” multi-processor targets. For example, we have classes named CGFullConnect and CGSharedBus that represent sets of homogenous single-processor targets of arbitrary type, connected in either a fully connected or shared-bus topology, with parametrized communication costs. We are also working on models for actual multiple-processor systems such as the CM-5, the AT&T DSP-3, the ordered transaction architecture [23], the Ariel Hydra board, and the Spectrum VASP.

The design of Ptolemy is also intended to support heterogenous multi-processor targets. For such targets, certain actors must be assigned to certain targets, and the cost of a given actor is in general a function of which child target it is assigned to. We have developed parallel schedulers that address this problem [17].

2.3 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 parallel 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.

2.3.1 Single-processor schedulers

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 [19].

2.3.2 Parallel scheduling

We have implemented three scheduling techniques that map SDF graphs onto multiple-processors with various interconnection topologies: Hu's level-based list scheduling, Sih's dynamic level scheduling [17], and Sih's declustering scheduling [18]. The target architecture is described by its Target object, a kind of MultiTarget. The Target class provides the scheduler with the necessary information on interprocessor communication to enable both scheduling and code synthesis. Targets also have parameters that allow the user to select the type of schedule, and (where appropriate) to experiment with the effect of altering the topology or the communication costs.

The scheduling techniques implemented in Ptolemy are retargettable in that they do not assume any limited set of interconnection topologies. When a scheduler incurs a requirement of interprocessor communication between processors, it instructs the target object to schedule the hardware resources for communication and compute the communication cost. The scheduler uses this information to decide whether the incurred communication cost is low enough to merit exploiting the parallelism. For example, in the OMA target [24], which has a shared-bus and shared-memory architecture, the requests for the shared bus from all processing elements are determined at compile-time. Taking advantage of the communication compile-time information, we can reduce the run time communication costs.

The multiple-processor scheduler produces a list of single processor schedules, copying them to the child targets. Given these single-processor schedules, the same schemes as discussed above are re-used to generate the code for each child processor target. Currently, we are targeting the Sproc from Star Semiconductor, the CM5 from Thinking Machines, the DSP-3 from AT&T, and various parallel machines using the Motorola 56000 and 96000 DSPs.

2.3.3 Extending beyond the SDF model: dynamic constructs

We have applied the idea of mixed-domain scheduling to support dynamic constructs for code generation. Here a dynamic construct is a data-dependent construct such as if-then-else, do-while, or recursion. Ptolemy defines a new domain called CGDDF for these dynamic constructs. By putting this new domain inside an "SDF Wormhole", the whole application can be scheduled statically. The dynamic constructs inside the SDF Wormhole change the run time execution profile from the scheduled one. We have developed a technique that schedules dataflow graphs with run-time decision making, aiming to minimize the cost of run-time decisions [14]. We will define a specific Target class for this dynamic construct domain and generate suitable control code for the target architecture corresponding to the dynamic constructs.

2.4 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. We also plan to implement the actors Spread and Collect (described in section 2.5.2) as AnyAsm stars. A BlackHole star is a data sink that discards its

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

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 [25]. 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 in [26] 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, code blocks, a start() method, an initCode() method, a go() method, a wrapup() method, and an execTime() method. Portholes, states and code blocks 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. Code blocks are a pseudo code specification of the target language. By pseudo code, we mean that the code block 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.

Start(), 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 start(), initCode(), go(), and wrapup() methods is when the method is called. The start() 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 initCode() method is called before the schedule is generated and after the memory is allocated; code generated by initCode() appears before the main loop.

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.

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 code blocks 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.

2.5 Interprocessor Communication

2.5.1 Send/Receive

When the target architecture is a multiple processor system, the programmer selects a parallel scheduler that best fits best the target and the application. The parallel scheduler determines which actors to assign to which processing elements, as well as when to execute them in each processing element. As an example, consider the simple case in figure 6, where all blocks are homogeneous (producing and consuming a single token).

Suppose that the scheduler generates the schedule as shown in the Gantt chart in figure 7. By assigning star B and star A to different processors, the parallel scheduler introduces interprocessor communication between processor 2 and processor 1. The cost of the communication overhead is dependent on the target. Based on the information that is specified in the target definition, the scheduler schedules the communication resources and reserves the time-slots in the generated schedule.

The next step is to generate code for each processor. For processor 2, code for star B and the "send" star should be generated sequentially. To generate code, however, it is not sufficient to concatenate the code of star B and the code of the "send" star. We first have to allocate the memory and registers appropriately in the processors. Since each processor is also a target, it can allocate the hardware resources suitably for the generated code, given a certain galaxy. Thus, sub-universes are generated for the individual processors after the parallel scheduling is performed, as shown in figure 8. Note that the "send" and "receive" stars are automatically inserted by the Ptolemy kernel when creating the sub-universes. The multiple-processor target class is responsible for defining "send" and "receive" stars.

Once the generated code is loaded, processors run autonomously. The synchronization protocol between processors is hardwired into the "send" and "receive" stars. One common approach in shared-memory architectures is the use of semaphores. Thus a typical synchronization protocol used is to have the send star set a flag signaling the completion of the data transfer; the receive star would then wait for the proper semaphores to be set. When the semaphores are set, the receive star will read the data and clear the semaphores. In a message passing architecture, the send star may form a message header to specify the source and destination processors. In this case, the receive star would decode the message by examining the message header. The routing path from the source to the destination processor is determined at the compile-time as explained in section 2.3.2. Any specific routing algorithm and routing mechanism are not assumed in Ptolemy, but rather should be provided by the target class.

2.5.2 Spread/Collect

In the example of figure 6, we assume that the graph is homogeneous: no sample rate change occurs in the blocks. In such homogeneous applications, each star is naturally assigned to one processor. However, many signal processing applications are multirate, allowing us to split the invocations of a star across multiple processors. Furthermore, operations on blocks of samples, such as an FFT, or operations on vectors make an SDF graph non-homogeneous. Consider the simple multirate application in figure 9, where block A generates two tokens and block B consumes three tokens. One iteration of this universe consists of three invocations of block A and two invocations of block B. The precedence relation among these invocations can be described with the acyclic precedence graph (APG) as shown in figure 10. We assume that there is no data dependency between invocations of block, and assume the same for block B. In the figure, A1 represents the first invocation of A, A2 represents the second, etc. The APG graph represents the communication pattern between the invocations.

In a target architecture with two processors, a valid schedule for the APEG graph is shown in figure 11. According to the schedule, the target would splice the appropriate send and receive stars into the graph. As can be seen in the APEG, B1 receives data from not only A1 but also A2. Also note that A2 has been assigned to the other processor. Since block B1 consumes 3 tokens (figure 10), we need a special block to collect tokens from sources A1 and A2 for the input to B1, and to preserve the appropriate order. This special block is called a *Collect* star. The sub-universe created for processor 1 is illustrated in figure 12-(a). The Collect star gathers the outputs from both block A and the receive star.

On the other hand, two invocations of block A are assigned to second processor. Among the four output tokens generated from block A in this processor, the first output token is routed to processor 1 and the rest are fed into block B. This behavior can be expressed by introducing another special block, called a *Spread* star, as shown in figure 12-(b). Note that the sample rate is changed between block A and the Spread star, causing block A to be executed twice. The Spread star directs the first output token of block A to the input buffer of the “send” star; the remaining three tokens are directed to the input buffer of block B.

If memory is used to communicate between blocks, then in most cases it is possible for the Collect and Spread to be implemented simply by overlaying memory buffers; in such cases no code is required to implement these blocks. The AnyAsm pseudo-domain described in section 2.4 provides facilities for actors that work by this kind of buffer address manipulation.

It is worth emphasizing that the sub-universe does not express the execution order of the blocks, which is already determined by the parallel scheduler. For example, in figure 12-(b), the execution order of this block is not A, A, Spread, “send”, and B, as might be expected if an SDF scheduling were to be performed with the sub-universe. The order is A, “send”, A, and B according to the schedule in figure 11. The sub-universes are created only for the allocation of memory and other resources before generating the code.

2.5.3 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 13-(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 13-(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 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.

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 14. 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 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 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 5.0). 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 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 objected 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.

5.0 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 [19] and dynamic constructs [14].

6.0 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 iteration*, Master's Thesis, University of California at Berkeley, 1991.

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

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

[21] 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.

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

[23] J.C. Bier and E.A. Lee, "A Class of Multiprocessor Architectures for Real-Time DSP," *International Symposium on Circuits and Systems*, vol. 4, IEEE, 1990, p. 2622-2625.

[24] S. Sriram and E.A. Lee, "Design and Implementation of an Ordered Memory Access Architecture," *International Conference on Acoustics, Speech and Signal Processing*, Minneapolis, MN, IEEE, 1993.

[25] 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.

[26] 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.

7.0 Biographies

José Luis Pino (pino@EECS.Berkeley.EDU) is a doctoral candidate in the Department of Electrical Engineering and Computer Sciences at U.C. Berkeley. His current research activities include automated code generation for DSP systems. In 1990, he received a B.E.E. from the Georgia Institute of Technology. While a student at Georgia Tech, he worked at General Dynamics in Fort Worth, Texas. Also as an undergraduate, he conducted research in the area of analog neural networks for which he received the Sigma Xi Research Award (1990) and the UROP Outstanding Research Award (1990).

In 1990, he was awarded an ONR Doctoral Fellowship, a Hertz Research Fellowship Grant, an AT&T Bell Laboratories CRFP Fellowship, and an NSF Fellowship. In the summer of 1990 he worked at AT&T Bell Laboratories in Murray Hill, New Jersey developing numerical root finding techniques for LPC root extraction. He received his Masters of Science in Electrical Engineering from U.C. Berkeley in the Spring of 1993.

Soonhoi Ha (sha@ohm.EECS.Berkeley.EDU) received the B.S. (honors) and the M.S. degrees in electrical and electronics engineering from Seoul National University, Seoul, Korea, in 1985 and 1987, respectively. He received the Ph.D. degree in the Electrical Engineering and Computer Science Department at U.C. Berkeley in 1992. He is currently a post doctoral research engineer in the same department at U.C.Berkeley.

He has been actively involved in the Ptolemy project since its birth. Ptolemy is a software environment that supports heterogeneous system specification, simulation, and design. He implemented the simulation kernel for dynamic dataflow graph and discrete event system, and the basic interface for mixed paradigm simulation. He also implemented several multiprocessor scheduling algorithms. His recent efforts are focused on generating C-codes for multiprocessor target architectures, workstations, and AT&T DSP3 machine, from the large grain data flow program representation. He is a member of IEEE.

Edward A. Lee (eal@EECS.Berkeley.EDU) research activities include parallel computation, architecture and software techniques for programmable DSPs, design environments for development of real-time software and hardware, and digital communication. He was a recipient of a 1987 NSF Presidential Young Investigator award, an IBM faculty development award, the 1986 Sakrison prize at U.C. Berkeley for the best thesis in Electrical Engineering, and a paper award from the IEEE Signal Processing Society. He is co-author of “Digital Communication”, with D. G. Messerschmitt, Kluwer Academic Press, 1988, and “Digital Signal Processing Experiments” with Alan Kamas, Prentice Hall, 1989, as well as numerous technical papers. His B.S. degree is from Yale University (1979), his masters (S.M.) from MIT (1981), and his Ph.D. from U.C. Berkeley (1986).

From 1979 to 1982 he was a member of technical staff at Bell Telephone Laboratories in Holmdel, New Jersey, in the Advanced Data Communications Laboratory, where he did extensive work with early programmable DSPs, and exploratory work in voiceband data modem techniques and simultaneous voice and data transmission. He is chairman of the VLSI Technical Committee of the Signal Processing Society, co-program chair of the 1992 Application Specific Array Processor Conference, a senior member of the IEEE, and on the editorial board of the *Journal on VLSI Signal Processing*.

Joseph Buck (jbuck@ohm.berkeley.edu) is a graduate student and Ph.D. candidate in the Electrical Engineering and Computer Sciences Department at U.C. Berkeley. His research interests include multiprocessor scheduling of real-time digital signal processing algorithms and design environments for simulation and software development. He received the B.E.E. degree from Catholic University of America (1978) and the M.S. in computer science from George Washington University (1981). From 1979 to 1984 he participated in research in speech coding, speech noise reduction, and speech recognition at the Naval Research Laboratory in Washington, DC.

From 1984 to 1989 he worked at Entropic Speech, Inc., Cupertino, CA, on real-time implementations of speech compression algorithms for telephony applications, as well as a widely used set of tools for digital signal processing research on Unix systems. Convinced that there had to be a better way of developing real-time digital signal processing software, and inspired by

Edward Lee's Gabriel project (see references), he returned to academia, where he is having considerably more fun.

8.0 Acknowledgments

The authors gratefully acknowledge the support of the Semiconductor Research Corporation, DARPA, AT&T Bell Labs, Office of Naval Research, Bell Northern Research, Motorola, and Rockwell.

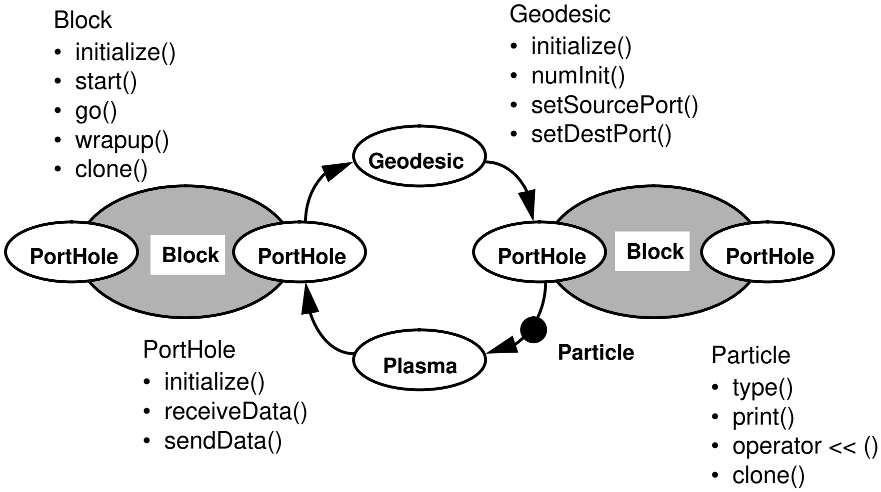

Figure 1. Block objects in code generation applications of Ptolemy synthesize code in some target language. PortHoles and Geodesics provide methods for managing the exchange of data between blocks.

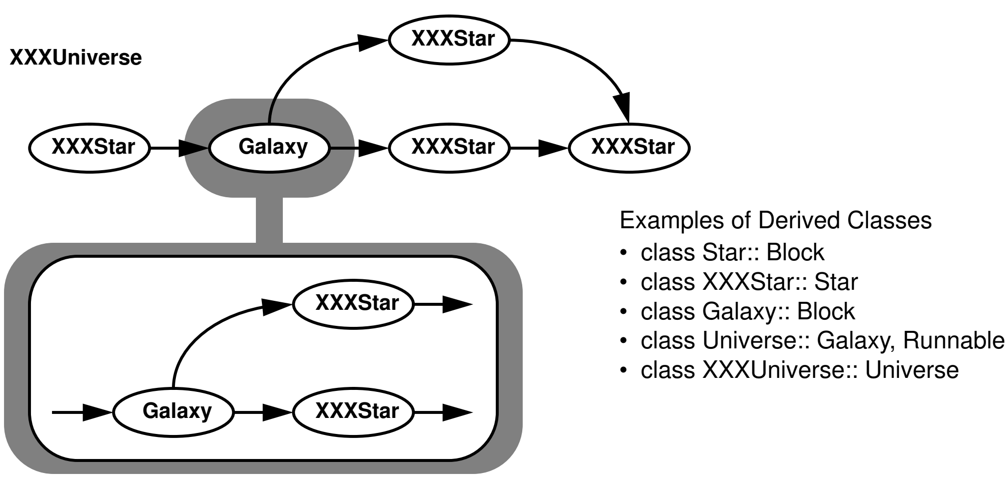

Figure 2. A complete Ptolemy application (a Universe) consists of a network of Blocks. Blocks may be Stars (atomic) or Galaxies (composite). The "XXX" prefix symbolizes a particular domain (or model of computation).

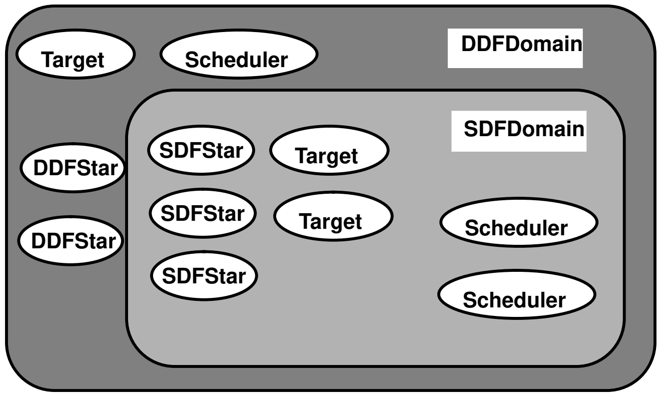

Figure 3. A Domain (XXX) consists of a set of Stars, Targets and Schedulers that support a particular model of computation. A sub-Domain (YYY) may support a more specialized model of computation.

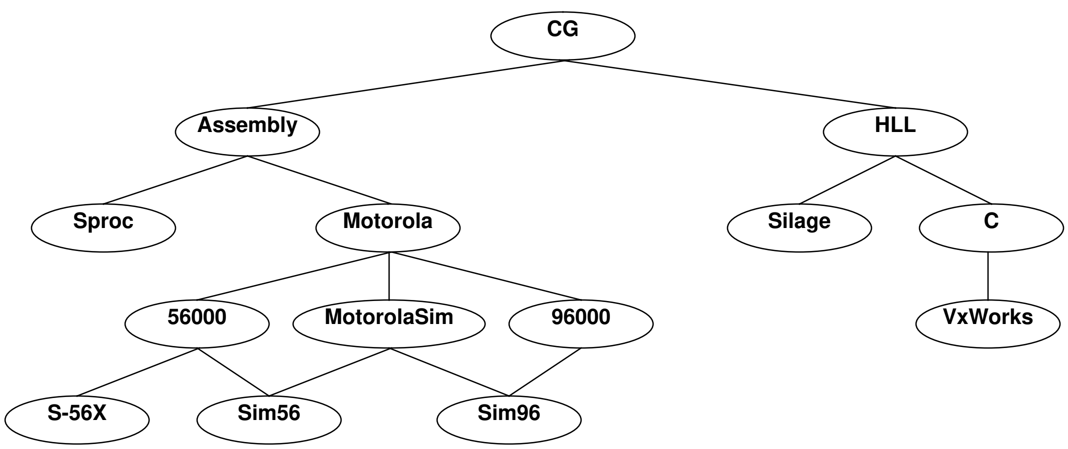

Figure 4. Inheritance Tree for Single Processor Targets.

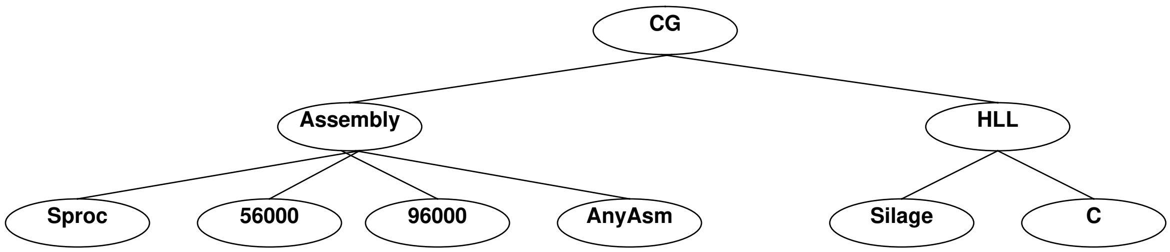

Figure 5. Inheritance Tree for Code Generation Stars.

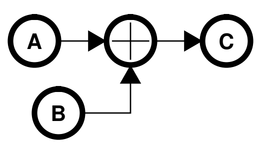

Figure 6. A simple example for illustrating the send/receive mechanism.

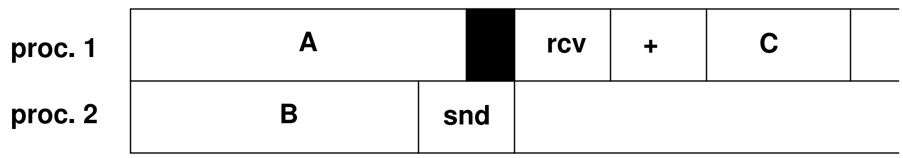

Figure 7. Scheduling result for the example in figure 6.

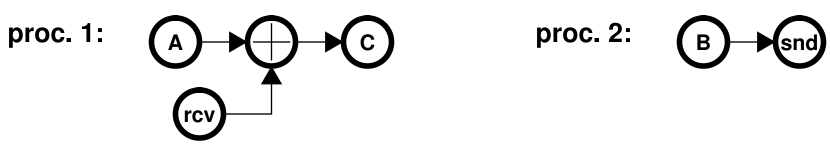

Figure 8. Sub-universes created for processing elements based on the scheduling result in figure 7.

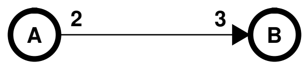

Figure 9. A multirate example for illustrating the spread/collect mechanism.

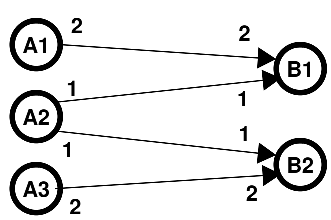

Figure 10. Acyclic precedence graph of the example in figure 9.

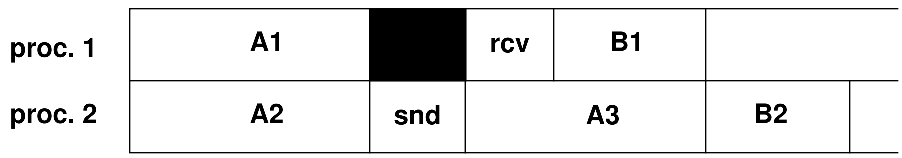

Figure 11. Scheduling result for the example in figure 9.

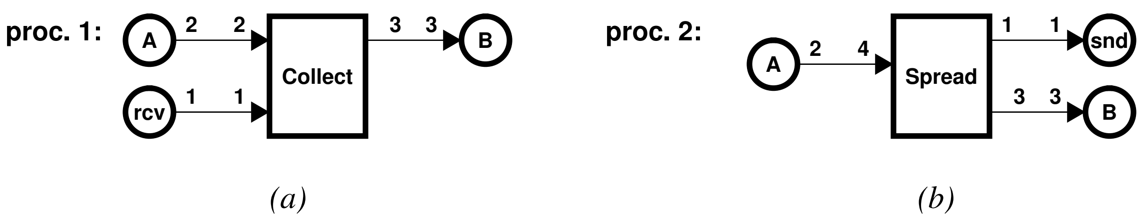

Figure 12. Sub-universes created for processing elements based on the schedule of figure 11. Special blocks, Spread and Collect, are used.

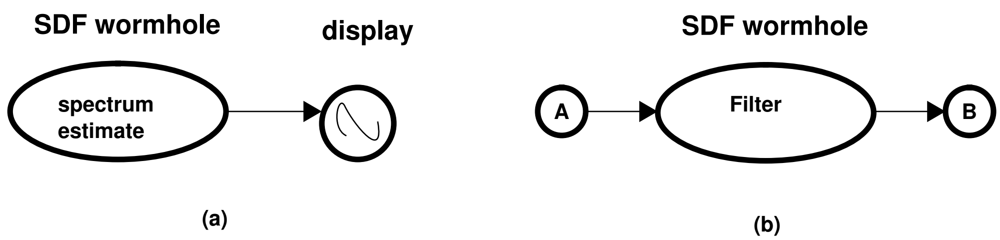

Figure 13. Examples of Host-to-DSP interaction using wormholes.

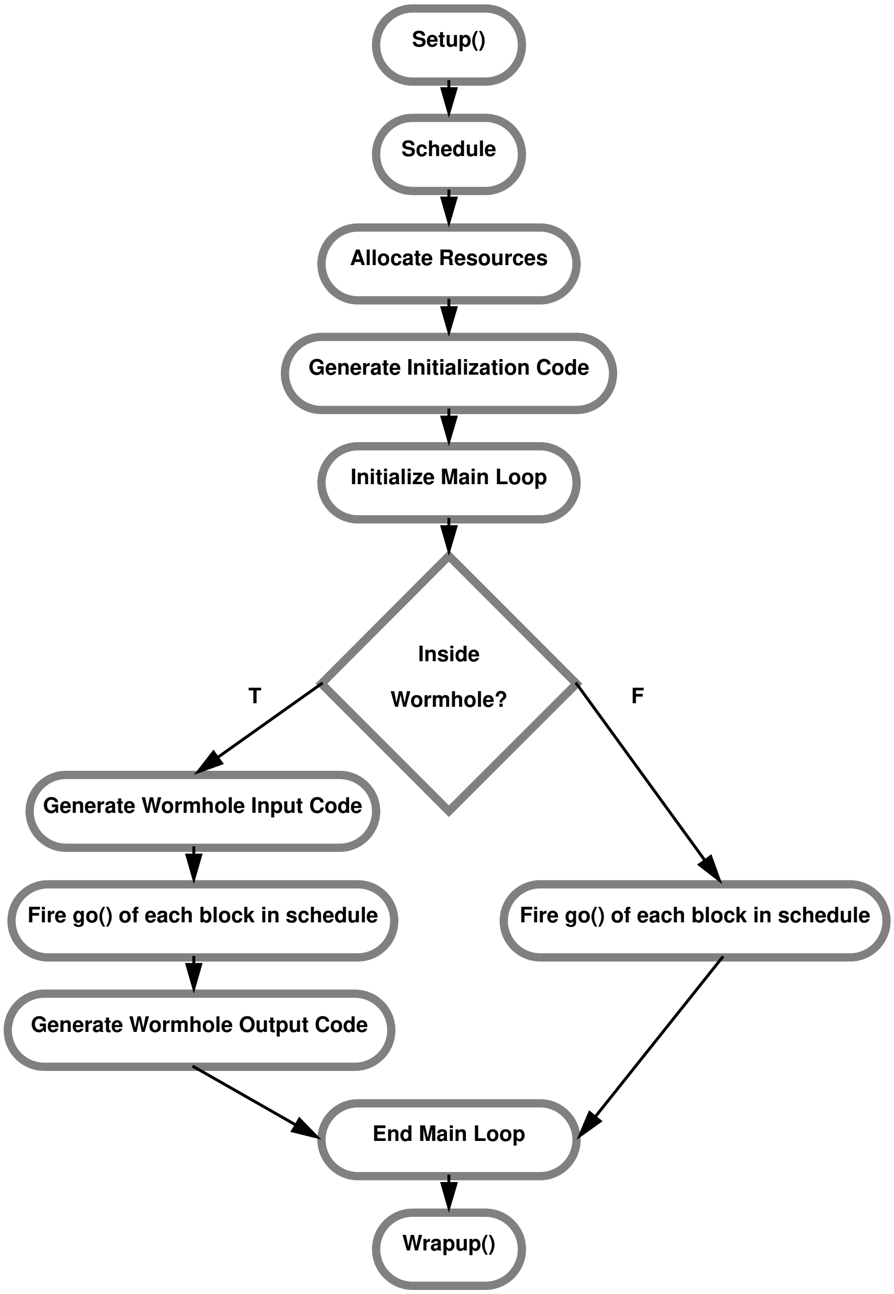

Figure 14. Code Generation Procedure