Code Generation for VSP software tool in Ptolemy

Sun-Inn Shih
MS Report (Plan II)

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720

1.0 Introduction

In the past few decades, the importance of video signal processing in the communication industry has expanded drastically. Simple broadcasting can no longer satisfy the demands of the public. As a result, multi-media has became one of the fastest growing business in the industry. Because of the large data processing involved, software implementation of video algorithms for real-time purpose is not realistic. However, hardware reconfiguration is not an easy task. The solution to this dilemma is programmable video processing chips.

In this project we extend the Ptolemy system developed in UCB to incorporate a programmable, extendable video processing chip called VSP. It allows rapid prototyping and evaluation for real time video applications. Currently, the software package that comes with the VSP chips includes a graphic editor, simulators, a code generator, a scheduler, and other related tools. A video algorithm is designed in the graphic editor and mapped to the hardware. The graphic editor uses signal flow graph (SFG) semantics where an algorithm is represented by nodes and arcs. A node represents a function chosen from a limited set of functions provided by the VSP package. An arc between two nodes represents data transfer between the nodes. The code generator in the VSP package generates the micro code for an algorithm specified by a SFG graph.

Ptolemy is a software environment that supports heterogenous system specification, simulation, and design. Ptolemy is programed in C++. The object-oriented framework allows diverse models of computation to co-exist and interact.

One of the computational models supported by Ptolemy, the Synchronous Data Flow model (SDF), is very similar to SFG semantics. In SDF, algorithms are also represented by nodes and arcs. Both semantics are capable of sample-rate conversion.

In this project, a VSP design environment, called VSP domain, is built in Ptolemy. It consists of two execution modes for a design. The first one simulates the design using SDF semantics. The second one generates the description of the algorithm in SFG. This description is called the design file [1]. The design file is written in the high level language used to communicate between different software tools, as provided by the VSP software package.

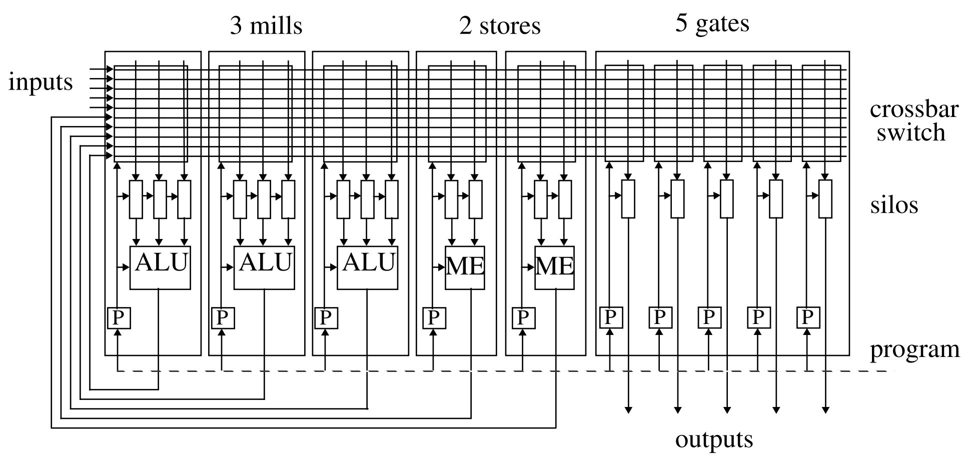
This implementation is a step toward real-time video processing in Ptolemy. It has two advantages over directly designing the algorithm in SFG. First of all, the SDF scheduler can automatically generate the execution period of each actor. This is manually set in SFG. Second, the heterogenous backbone of Ptolemy is potentially capable of simulating interactions between a video system and other components in the communication network. Although this is not been implemented in this project, the project serves as an initial step.

The organization of the report is as follows. Section 2 is an introduction to the VSP system. Section 3 gives a brief overview of Ptolemy. By relating a parameter in SFG called *cycletime* and a parameter in SDF called *repetition rate*, we can tie the two semantics together. Section 4 explains how SFG is generated from SDF by drawing the relationship between the two parameters. Section 5 presents VSP domain and the two execution modes. Examples are given in Section 6. Section 7 concludes the report.

2.0 VSP and its software tool

VSP stands for video signal processor. A VSP contains ten pipelined processing elements (PEs) operating in parallel at 27 MHz. The architecture of a VSP is shown in figure 1. A PE can be a mill, a store, or a gate. A mill consists of an arithmetic and logic element (ALE), three silos, and a section of the crossbar switch. A store has a memory element(ME), two silos, and a section of the crossbar switch. The gate only has one silo, and a section of the crossbar switch. Each PE has its own set of programs (see the lower left corner of each PE). The program controls either the ALE or the ME, the silos, and the crossbar switch belonging to the PE. The program, consisting of a small fixed number of unconditional instructions, is periodically executed. At each clock cycle a PE starts a new instruction in the program. The length of the program is the period between two successive executions of the same instruction line in the program.

Figure 1. VSP architecture



The VSP has five inputs and five outputs. The inputs and outputs of PEs along with the external inputs are connected to a crossbar switch with 18 outputs. Each output of the crossbar switch can be programmed to select one of the ten inputs of the crossbar switch. The silos at each output of the crossbar switch are capable of implementing delays of 1 to 31 clock cycles. The inputs and the outputs of a VSP chip can be connected to the external components to form an application or can be connected to other VSP chips. A VSP network is a system that consists of a number of interconnected VSPs. For the format of the inputs and the outputs of a VSP, refer to [1].

There are ten independent programs in each VSP chip; programming and debugging task for this VLIW architecture are difficult and tedious. The software tools provided by Philips are designed to make this more manageable.

The manual for the VSP software tools provides three major steps as guidelines to program the VSP. First graphically edit the algorithm using the SFG environment in the VSP graphic editor. Second, map the SFG onto a VSP network. Third, generate the microcode for VSP hardware.

A high level language is used to communicate between the software tools. The file that describes the design is called a design file. Two simulation tools are provided by Philips for debugging purposes. The simulation software provided in the VSP package reads design files. One of the simulation tools provided is for functional simulation. The next section briefly discusses the concept of SFG. The subsection after that explains the design file. The last subsection discusses how *cycletime* is assigned.

2.1 The VSP graphical language, concept

In SFG semantics, a video algorithm is described as a set of nodes interconnected by arcs. A node in SFG is called an **operation** and an arc is called an **operand**. The function specified by an operation is executed at the period proportional to the *cycletime* attribute of the operation. The system is specified for all iterations. In addition to operands and operations, shared memory is introduced by a structure called **common**. For example, a memory operation has an attribute to specify the common they are link to. Two memory operations that link to the same common share the same memory space. The initial content of a common is defined by a **table**.

2.1.1 Cycletime definition

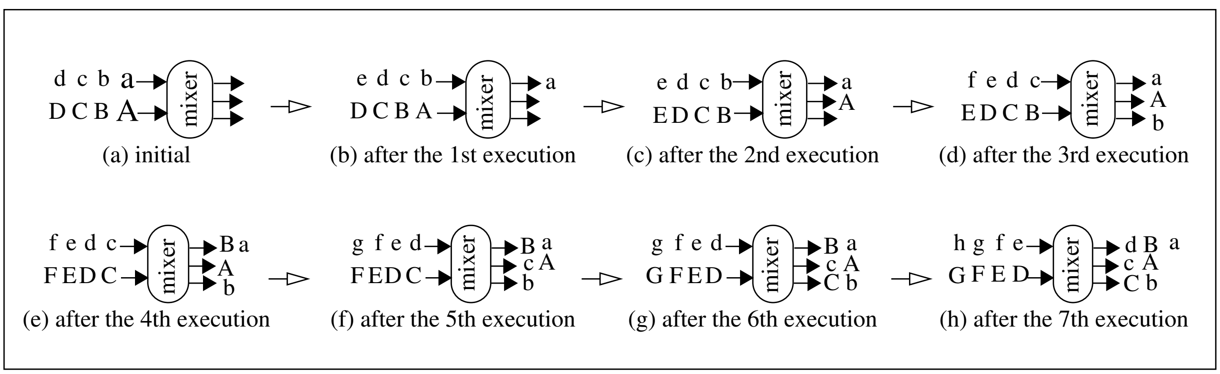
All operations require the *cycletime* attribute. This attribute is an integer that defines the time between consecutive executions of an operation. This is given as a multiple of the clock cycle, which is 1/(27MHz). For example, an operation with cycletime 2 is executed at the frequency of 13.5 MHz and an operation with cycletime 5 is executed at the frequency of 27/5 MHz. The users are required to defined this attribute manually for each operation in their algorithms.

2.1.2 Operation

There are five major types of operation:

- ***an alu*** operation performs arithmetic and logic functions. It has three data inputs and one data output. This is closely related to the mill structure in the VSP hardware. It is capable of performing add, subtract, logic functions, and comparison. The exact function performed depends on an attribute in the operation and the data value of the third input. This allows limited data dependent calculation.
- ***a mem*** operation can read from, or write to an address space defined in the operation attributes. If it is a write operation, it has a data input and an address input. If it is a read operation, it has an address input and a data output. A mem operation almost always links to a share memory space (common). In addition, mem operations usually exist in pairs. The execution order of a group of mem operations is indicated by a special type of operand called *No_value* operand. A more detail discussion of *No_value* operand is in section 2.1.3.

Figure 2. execution of a mixer operation with two inputs and three outputs



- ***a mixer*** operation allows sample rate conversion in the algorithm. This is the operation that supports multi-rate in SFG semantics. The *inputs* and *outputs* attributes specify the number of input arcs, and output arcs respectively. If the *cycletime* of a mixer is α then its input sample period must be α * *inputs* and its output sample period must be α * *outputs*. The mixer interlaces the inputs into an internal stream of samples and then unravels it to the outputs. For example, figure 2 shows the execution of a mixer with two inputs and three outputs. It is important to note that at each execution, only one token is consumed and only one token is produced. Therefore, a sample is consumed at the first input arc every 2* α clockcycles and a sample is produced at the first output arc every 3 * α clockcyles.

- ***an interm*** operation is an input operation. It has one output. The *filename* attribute specifies the file that the algorithm reads from during the simulation.
- ***an outterm*** operation is an output operation. It has one input. During simulation it writes its input to the file specified by the *filename.*

2.1.3 Operand

An operand connects an output of an operation (call it the from-operation) to an input of an operation (call it the to-operation). It has a direction associated with it. The direction indicates the signal sample flow. The operand can't change the signal sample rate, which implies that the signal sample rate is constant over the operand. An operand has three attributes. The *comment* attribute is for user comments. *shift* specifies the bit shift needed to be performed on the value from the from-operation. The *constant* attribute is a constant added to the shifted value prior forwarding it to the to-operation.

It is possible for an operand to have no from-operation. In this case, the value of *constant* is continuously fed into the to-operation.

The *No_value* attribute in the operand statement specified whether or not this operand is a *No_value* operand. Its presence is to indicate precedence information. In another words, the *No_value* operand requires that the from-operation (or the previous iterations of the from-operation if *delay* exceed one) has to finish before the to-operation can start.

2.1.4 Common and Table

Common links mem operation to a shared memory structure. From the hardware configuration in figure 1, there are two memory elements for each VSP chip. When a mem operation is linked to a common, its accessing address space is indicated by the first address and the length of the address space, namely *first_address* and *length* attributes of the mem operation.

A shared memory space can have initial value. A table is a list of numbers. When a common is created, *fill_table* attribute contains the name of the table that have the initial values. *fill_address* attribute is the starting address of this initial filling.

2.2 Design file

The design file is a high level texture language used by the VSP software tools to describe the system. A design file can be generated from the graphical editor of the VSP package. It can be modified by the scheduler. When fed into the simulators provided by VSP package, it is simulated. When it is fed into the code generator provided by the VSP software package, micro code for the VSP hardware is generated.

The design file states the operations and describes the topology of the algorithm by operand statements. There are special statements for commons and tables. For detail of this file format, refer to [1].

One goal of this project is to directly generate this texture description from a design in Ptolemy.

2.3 Cycletime assignment

The *cycletime* attribute of each operation is assigned by the users manually. An incorrect *cycletime* assignment will result in error message from the tools in the VSP package. There are three guidelines to assign correct *cycletime*.

- Except for the mixer operation, the *cycletime* of an operation must match the sample period of the input(s) and output(s) of the operation. In other words, only the mixer is capable of sample rate changes.

  cycletime = input sample period = output sample period
- Given a mixer of *cycletime* α, its input sample period has to equal to *inputs*$*\alpha$ clockcycles and its output sample period has to equal to *outputs*$*\alpha$ clockcyles. (See section 2.1).

  cycletime (of a mixer) = input sample period/ *inputs*
  = output sample period/ *outputs*
- The sample rate does not change over an operand.

By obeying these rules, the user can traverse the SFG and correctly assign the *cycletimes* of each operation. However, the assignment can be done automatically. In fact, *cycletime* is readily derived from the *repetition rate* calculated in SDF semantics. The following sections will define the *repetition rate* (sec 3.1) and explain this procedure (sec 4).

3.0 Overview of Ptolemy

An algorithm in Ptolemy is represented as a dataflow graph constructed by interconnecting both user-created and existing library blocks. The fundamental building blocks in Ptolemy are called **stars**; they contain code segments for either code generation or execution. Data passes between blocks in discrete units called **particle**s. A particle can be either as simple as an integer value or as complicated as an image frame. Arcs between blocks specify the data path. Furthermore, hierarchy is introduced to manage complex systems. In Ptolemy, hierarchy is achieved through an intermediate class of blocks called **galaxies**. Each galaxy is composed of a subsystem of interconnected galaxies and stars. Each galaxy is associated with a **domain**, which specifies the computational model of the galaxy. The user can choose a **target** from the list of available targets in the domain. A target defines the mechanism by which a system is executed. Every target is associated with a **scheduler**, which determines the operational order of each block in the application. Finally, a complete application in Ptolemy is called a **universe**.

Next subsection is a brief discussion of the SDF domain, on top of which our implementation is build. For details on SDF, refer to [3, 4, 6].

3.1 Brief introduction to SDF semantics

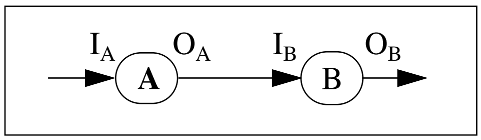
SDF stands for synchronous dataflow. SDF semantics regard an arc as a stream of data. This semantics is characterized by the restriction that the number of tokens produced and consumed by a star at each execution is fixed at compile time. The tokens on each arc form an ordered sequence. The SDF scheduler relies heavily on balancing the consumption and production of tokens on the arcs. This keeps the buffer size on the arcs bounded, and the system is capable of processing data streams of infinite length without overflowing the buffers on the arcs.

To balance the number of tokens at the both ends of the arc in figure 3, we need to relate the number of executions of A (denoted by r_A) to the number of executions of B (denoted by r_B). I_A and O_A denote the number of tokens produced and consumed by A at each firing. Similarly, I_B and O_B denote the number of tokens produced and consumed by B at each firing. The constraint:

$$r_A O_A = r_B I_B$$

is called the *balance equation*.

Figure 3.

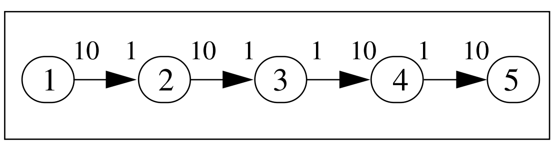


Given a graph, a set of balance equations are collected into a matrix form Γ and the smallest positive solution for **r** where **r** satisfied:

$$\Gamma r = 0$$
$$r > 0$$

is the repetition vector. Γ is the matrix of the coefficients of the r_i's in the balance equations. In Γ, each column describes a particular star and each row describes a particular arc. The repetition rate of star A is r_A.

Figure 4. example



A simple example of Γ and **r** is shown in figure 4, where Γ is as follows:

$$\Gamma = \begin{bmatrix} 10 & -1 & 0 & 0 & 0 \\ 0 & 10 & -1 & 0 & 0 \\ 0 & 0 & 1 & -10 & 0 \\ 0 & 0 & 0 & 1 & -10 \end{bmatrix}$$

In this example, the second column of Γ indicates that the second star receives one token from the first arc and sends ten tokens to the second arc at each execution. The second row indicates that at every firing of the second star, the second arc receives ten tokens. In addition, at every firing of the third star, the second arc sends one token to it. The repetition rates of the stars are r_1=1, r_2=10, r_3=100, r_4=10, and r_5=1.

Currently, there is no timing information associated with our calculation. We can introduce time into this calculation by restricting the executions (firings) of the same star to be periodic with a fixed period. Assume a system with k stars' *repetition rates*: $\mathbf{r}^T = [r_1\ r_2 \ldots r_k]$, and firing periods: $\mathbf{p}^T = [p_1\ p_2 \ldots p_k]$. r_i's and p_i's has to satisfy $\boldsymbol{w} = r_i{*}p_i$, for all i, where $\boldsymbol{w}$ is some positive number. p_j is solved for:

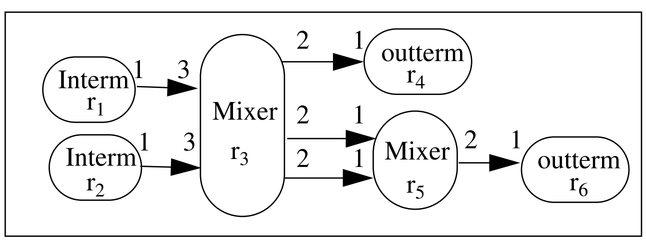
$$pj = \frac{(lcm\{r_i \mid \forall i\})}{rj} \cdot p_b$$

p_b is any arbitrary positive number.

It is important to note that p_b is an extra degree of freedom in calculating the firing period of the stars.

Figure 5 is an example in SDF semantics.

Figure 5. SDF semantics



$\mathbf{r}^T$ = [3 3 1 2 2 4] and $\mathbf{p}^T$ = [4 4 12 6 6 3].

From the above example, it is clear that SDF supports multi-rate by allowing stars to consume and produce multiple tokens at each firing. The SFG on the other hand uses the mixer star to do rate conversions. Section four will relate the two mechanism.

3.2 Target

In this project, a new domain called VSP domain is implemented. It is an extension of SDF domain in Ptolemy. Along with the domain is a library of SDF semantics stars. Target is used in Ptolemy to provide an additional layer on top of the domains. This mechanism allows us to build two run modes in this project. After interconnecting stars from the library, the user can select two run modes: simulation mode and code generation mode. Simulation mode is designed to do functional simulation that is equivalent to the one provided in VSP package, while code generation mode is selected for design file generation. Both these modes are implemented in **VSPTarget**.

4.0 From repetition rate to cycletime

The similarity between the SFG in VSP and the SDF semantics in Ptolemy is easily identified. For most operations in the VSP, the number of tokens consumed and produced by an operation is fixed. This is precisely the SDF model. Assume we have an universe with $\boldsymbol{k}$ stars and none of them is a mixer. Let the *repetition rate*s of the stars be $\mathbf{r}^T = [r_1\ r_2 ...\ r_k]$. The firing period is $p^T=[c_1*p_b\ c_2*p_b ...\ c_k*p_b]$, where $c_j = \boldsymbol{lcm}\{r_i|\ 0 < i < (\boldsymbol{k}+1)\}/r_j$ We can utilize the extra degree of freedom discussed in subsection 3.1 by choosing p_b. In this implementation, p_b is called ***SlowDown,*** where ***SlowDown*** is any positive integer. Finally, *cycletime* for the stars equals the firing period. The calculation for mixers are slightly more involved because it is not an SDF semantic block.

SDF requires all inputs of a star to be available at every firing of the star. A mixer does not have such requirement. This is precisely the reason why the input and output sample periods can be different from the *cycletime* of the mixer. A mixer accesses only one of its input arcs and one of its output arcs at each firing; the other input and output arcs are idle. For functional simulation purpose, such fine grain simulation in Ptolemy is not necessary.

The concept of the VSPMixer star is to process a set of data at each firing. For each input arc, the number of token consumed at one firing is *outs*. At each output arc, one firing produces *ins* tokens. (*ins* and *outs* are the equivalent of the *inputs* and *outputs* parameters in SFG mixer operations; they specify the numbers of input and output ports.) The total number of tokens in a set is *ins* * *outs*. At each firing, VSPMixer interlaces the inputs into an internal list of samples and then unravels it to the output. The best illustration for VSPMixer is in figure 2. The SFG semantics takes six firings from (a) to (g), the VSPMixer takes only one firing. At each firing of the VSPMixer, six tokens are consumed, three from each input arc. In addition, at each firing, six tokens are produced, two at each output arc. A more detail description of VSPMixer is in Section 5.4.

Assume the firing period for a mixer star is δ. The SFG mixer operation needs *ins* * *outs* firings to process a set of data, while the equivalent VSPMixer needs only one firing. Therefore, *cycletime* is $\delta/(ins * outs)$.

For the example in Figure 5, the *cycletime* is:

$$cycletime = \begin{bmatrix} 4 \\ 4 \\ 2 \\ 6 \\ 3 \\ 3 \end{bmatrix}$$

Note that the Mixer stars has $p_3 = 12$ and $p_5 = 6$. These numbers are divided by 2*3, and 2*1, respectively. *cycletimes* are 2 and 3 instead.

5.0 VSP domain

VSP domain is an extension of the SDF domain in Ptolemy. SDF semantics has no side effects. However, the shared memory accessing of SFG mem operation does permanent changes to the overall system. VSP domain supports this by a C++ class `Common`, which simulates shared memory space in SFG. This structure can be created and manipulated by VSPTarget The content of a `Common` object can be accessed by VSPMem, which is equivalent to SFG mem operation. `Common` is explained in Section 5.1

There are four basic components in a VSP network: operations, operands, commons, and tables. The star library has been constructed to supply the basic stars to build a VSP network in Ptolemy. The star library is discussed in section 5.2

VSPTarget has two run modes: simulation mode and code generation mode. Before running any run mode, VSPTarget runs the SDF scheduler. The scheduler calculates the *repetition rates* of the stars and sets up the firing schedule of the stars. The simulation mode simulates the SDF graph in the VSP domain. The code generation mode generates the design file for the algorithm. Section 5.3 and 5.4 talks about the two run modes in **VSPTarget**.

5.1 Common

`Common` is a C++ class that to simulates shared memory in SFG. A `Common` object is created by **VSPTarget** with a name and initial values. The data in the shared memory space is saved in a private fix array member `data[512]`, where 512 is the shared memory size of SFG. `Common` has two important public member functions: `read(int address)` and `write(int address, fix val)`. These functions manipulate the content of `data[]`. `read()` returns the data value at the address indicated while `write()` does not return anything.

`CommonList` and `CommonIter` are classes created to maintain the list of all the commons in the universe. `CommonList` is a list of commons. When a new common is created by **VSPTarget**, it is appends the new `Common` object to a `CommonList`. `CommonIter` is an iterater that iterates through `CommonList` to search a certain common by its name.

5.2 Star Library for VSP

There are seven major classes of stars in the Ptolemy VSP star library. VSPInterm, VSPOutterm, VSPAlu, VSPMem, VSPMixer, VSPShift, and VSPConstant. Most of them closely resemble the types of operation available in SFG. Since most of the SFG operations are equivalently SDF blocks, they can be easily implemented in Ptolemy. However, arcs in SDF semantics do not support functions provided by the operand type of the SFG. To solve this problem, the operands are promoted to stars when necessary. VSPShift stars represent operands with *shift*, and *constant* attributes; and VSPConstant stars represent operands with no from-operation. (Refer to section 2.1.3) In the following section, we briefly discuss the implementation of each star.

5.2.1 VSPInterm and VSPOutterm

parameters: filename (string)

inputs: VSPOutterm - i_1

outputs: VSPInterm - o_1

VSPInterm reads in a data file and VSPOutterm writes input data to a file. VSPInterm opens a file when the simulation starts. At each firing, it reads an integer from the file and produces a 12 bit fixed-point data sample at its output arc. VSPOutterm, on the other hand, reads a 12 bit fixed point data sample from its input arc and writes an integer to the file.

5.2.2 VSPAlu

parameters: instruction_type(string), inst_alternative (int)

inputs: i_0, i_1, control

outputs: o_1

VSPAlu simulates the arithmetic and logic functions provided by SFG alu operation. VSPAlu requires the user to specify an *instruction_type* and an *option*. The *instruction_type* includes add, sub (subtract), cmp (compare), and log (logic). Each of them has couple of variations which are specified by *option*. At each firing of the star, the proper function is executed according to the two parameters and the third input arc (*control*).

5.2.3 VSPMem

parameters: common_ref (string), first_addr (int), length (int)

VSPMem simulates memory access provided by the mem operation in VSP chips. There are a total of four VSPMem stars. The parameters that are shared by VSPMem stars are *common_ref, first_addr, length*. The *common_ref* indicates which `Common` object this star is associated with. **VSPTarget** is responsible for creating and maintaining the structure. When a VSP-

Mem is fired, it accesses the `CommonIter` in **VSPTarget** and looks for the `Common` object with the matching name *common_ref*. Obtaining the **Common** object, the VSPMem can read from or write to the object. *first_addr* and *length* specify the address space in the object that is accessible by the VSPMem star.

In SFG semantics, depending on whether or not a *No_value* (section 2.1.3) operand is connected to the mem operation, it can have different numbers of input and output arcs. The implementation enumerates the possible mem operations: VSPMemR, VSPMemW, VSPMemRNVO, VSPMemWNVO. R and W stand for Read and Write. NVO stands for No-Value Operand. Since the stars can be invoked when all the inputs of the stars are available, the presence of an arc connecting two stars impose an ordering of the firing. This is exactly the purpose of an *No_value* operand. Therefore, the simulation of an *No_value* operand is accomplished by forking a VSP-Mem star output arc to a VSPMemRNVO or VSPMemWNVO *no_value* arc.

All inputs and outputs of VSPMem stars are 12 bit fixed data type.

- VSPMemR/VSPMemW

  inputs: VSPMemR - i_0; VSPMemW - i_0, i_1

  outputs: VSPMemR - o_0; VSPMemW - o_0

  These are the most basic VSPMem. VSPMemR has an address input and an data output. VSP-MemW has two inputs, an address input and a data input. In the case where the user wants to express precedent, the output of these basic stars are connect with the *no_value* input of the The following two VSPMem stars.

- VSPMemRNVO/VSPMemWNVO

  inputs: VSPMemRNVO - i_0, no_value; VSPMemWNVO - i_0, i_1, no_value

  outputs: VSPMemRNVO - o_0

  This is case that the star at the other end of the *no_value* has to fire before the star can fire.

5.2.4 VSPMixer

inputs: i_0,..., i_ins

outputs: o_0,... o_outs

VSPMixer reorders the samples as in the SFG mixer operation. It implements a sample rate change. As mentioned in section 4, an VSPMixer processes data as a set. Each set consists of *ins*outs* data samples. A VSPMixer star, with *ins* input arcs and *outs* output arcs, requires *outs* tokens from each input arc at each firing. It produces *ins* tokens at each output arc. A VSPMixer is a commutator with *ins* input arcs followed by a distributor with *outs* output arcs. Figure 6a is a commutator with ***k*** input arcs. The elements of the input arcs are interlaced into a stream. Figure 6b is a distributor with ***k*** output arcs. A stream ***s*** is distributed orderly onto the output arcs. VSP-Mixer combines these two stars into one star.

Figure 6.

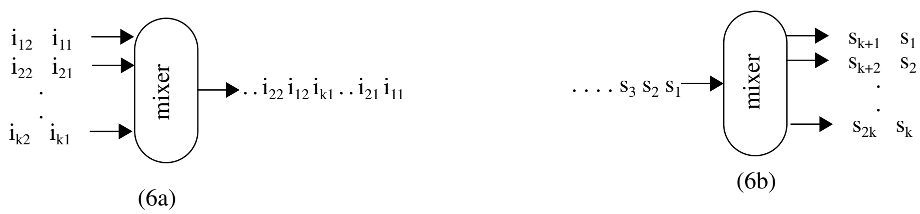
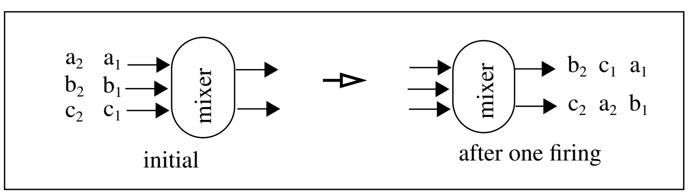


Figure 7 shows one firing of a mixer with three input arcs and two output arcs. Generalization of other combinations of *ins* and *outs* is straight forward.

Figure 7. execution of a VSPMixer with 3 inputs and 2 outputs



5.2.5 VSPShift and VSPConstant

parameters: VSPShift - shift (int), delay (int), constant (int);

VSPConstant - constant (int)

During code generation, VSPShift and VSPConstant are regarded as arcs with parameters. VSPShift simulates a VSP operand with *shift*, and *constant* attributes. VSPConstant simulates an operand with no from-operation (see section 2.1.3). In the simulation phase, VSPConstant is a source star that produces tokens with values equal to the *constant* parameter.

5.3 Simulation

In the simulation mode, **VSPTarget** first creates all the shared memory structure in the universe. It iterates through all the universe parameter looking for parameters with prefix "common". Whenever it encounters one, it creates a `Common` object and appends it to the end of its `CommonList`. The full name of the parameter becomes the name of the `Common` object. The content of the parameter is the name of the table for initial filling of the common (refer to section 2.1.4). For example, assume we have a common with name *commonxxx* and content "tableyyy". Then content of the universe parameter *tableyyy* is the data for initial filling of the common. In addition, the content of the universe parameter *fillxxx* indicates the starting address of the initial filling. This process is repeated till the end of the parameter list.

Finally, **VSPTarget** uses the SDF scheduler to determine the firing order of the star. When the star is fired, it executes the functions provided in each star.

5.4 Code Generation

In code generation mode **VSPTarget** never fires stars. Its purpose is to describe the graph by generating the design file. The target of a universe has access to each of the stars belonging to the universe. The target also has access to the parameters of the stars and that of the universe. **VSPTarget** accesses these parameters to generate the design file of a universe. **VSPTarget** makes a total of three passes through the star list.

- First pass: The target calculates the ***lcm*** (least common multiple) of *repetition rates* of all stars belonging to the universe, which have been computed by running the SDF Scheduler.
- Second pass: The target generates the operations corresponding to the stars.
- Third pass: The target generates the operands of the application.

When the design file is generated **VSPTarget** invokes the VSP graphic editor and displays the algorithm in the editor.

5.4.1 Common and Table

This is best illustrated by an example. Assume three universe parameters are defined: *commonxxx*, *fillxxx*, *tableyyy*. Let @ indicate the content of a universe parameter. In addition, assume @*commonxxx* = "tableyyy". VSPTarget generates two line of code from these[1]:

(table tableyyy "" length(@*tableyyy*) (@*tableyyy*))

(common commonxxx "" x1 y1 x2 y2 x3 y3 tableyyy @*fillxxx* false)

length(.) is the length of the (.). xi and yi are integers used to indicate the position of the common. The format is described in [1].

5.4.2 Operation (The Second pass)

The firing periods of the stars are calculated by dividing the ***lcm*** by the repetition rates of the stars. *cycletime* is calculated by the firing period times ***SlowDown*** and divided by the *ins* and *outs* parameters of the stars. (refer to section 4) If a star does not have these parameters, the parameters are assume to be 1. ***SlowDown*** is defined by the user as an universe parameter.

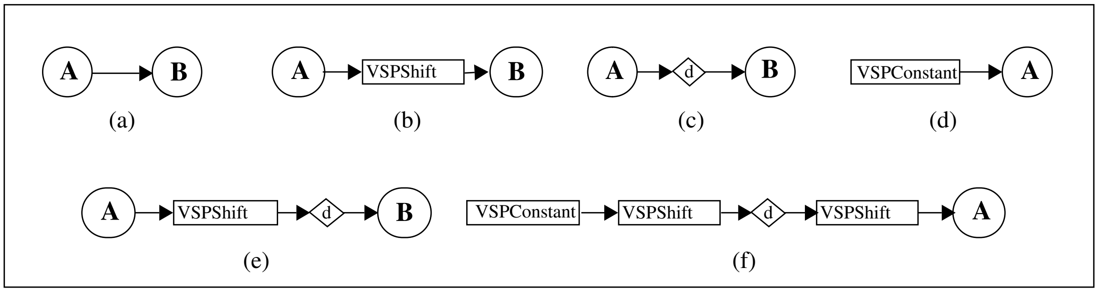
5.4.3 Operand (The Third Pass)

VSPTarget identifies each arc in order to generate the operand statement. For ease in later discussion, stars that are not VSPConstant or VSPShift are called proper stars. An operand in SFG can be represented by any of the following structures:

- An arc between two proper stars
- An arc with any number of delays between two proper stars
- Any number of VSPShift star(s) plus any number of arcs need to connect between two proper stars.
- VSPConstant and any number of VSPShift, delays, and arcs necessary to connect them to the input of an proper star.

Figure 5 shows some of the possible cases. The counterpart of a proper star in SFG semantics is an operation. In figure 7, **A** and **B** are proper stars, and the diamond shape block stands for delays.

Figure 8. Some cases that are equivalent to SFG operands



Two important characteristics are used to trace the topology.

- Every input arc of a star has one and only one corresponding star at the other end of the arc.
- An SDFShift star has one input arc and one output arc.

The **VSPTarget** iterates through all the proper stars. For every input arc in a proper star, **VSPTarget** traces it to the star at the other end of the arc. If a proper star is encountered an arc is identified. **VSPTarget** keeps count of the number of delays on the arc. If it is an vspShift star at the other end, **VSPTarget** finds the input arc of the star and keeps tracing until it encounters either a proper star or an VSPConstant star. Along the way, the **VSPTarget** adds up the constants, the delays, and the shift values. If an VSPConstant star is found, the operand has no from-operation.

6.0 Application examples

Two examples are described in this section. The first one is image subsampling by a factor of two. In image processing, an image is represented by a matrix of pixels. In a black and white image, a pixel has one value which specifies the intensity. A row across an image is called a scanline. This subsampling example discards every other scanline and every other pixel in the remaining scanlines. Therefore, after subsampling, the final image is one fourth of the original size. The procedure is frequently used in subband coding.

The second example is a simplified version of edge detection. An image is passed through a two dimensional filter that is a gradient based edge detection filter. The resulting data is compared to a threshold, and the pixel is considered part of an edge if the data is over the threshold.

Edge detection is used in feature extraction, and filtering is widely used in noise reduction, image enhancement and image coding.

The next two sections discussed the implementation of the examples.

6.1 Subsampling

Subsampling is frequently used in subband coding. In this example, a 128 * 128 grey scale image is the result of subsampling a 256 * 256 grey scale image.

6.1.1 Approach

There are two different types of operations involved: discarding every other sample in a scanline and discarding every other scanline. An VSPInterm star reads an image from a file. The file includes the sample rate of the input and a stream of rasterized pixel values of the image. A mixer with one input and two outputs discards every other sample (Refer to figure 9). We are only interested in the first output of the mixer. We choose the image width to be an even number so that we can safely discard the second output. The result of the mixer is written to a memory location.

Two memory accessing operations are used to discard every other scanline. After every write at location ***y*** is finished, a read is performed at location ***y***/2. We implement a modulo-256 counter, and a secondary counter derived from it that is half of the value of the primary counter. The primary counter is fed to the address input of the memory write operation, and the secondary counter is fed to the address input of the memory read operation. Notice in figure 9 we connect an VSPShift output to the input of VSPAlu instead of simply perform a shift to the data. This is because VSP can only perform shift at an input or output of an Alu operation. The resulting range of the write addresses is 0 to 255, and the address range for the read is 0 to 127. When the write operation writes two scanelines, the read operation reads only from the first scaneline by reading each pixel twice. After the read, ia mixer discards the redundant data.

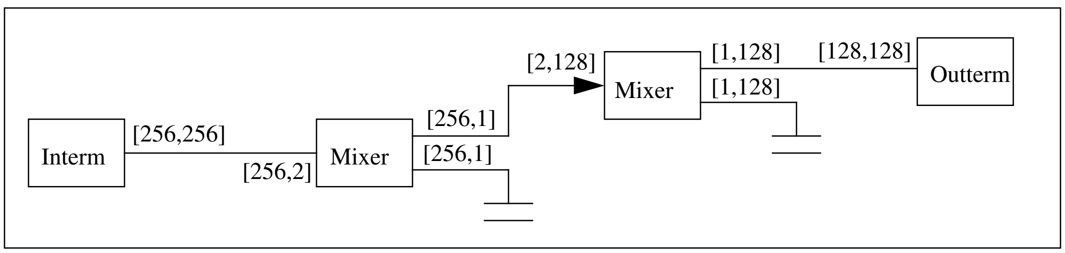
CGVSPTarget generates the equivalent graph in VSP software as shown in Fig. 10.

6.1.2 Discussion

This is an algorithm for subsampling if we are confined to subsample inside a VSP chip. It seems extremely complicated for this simple task. The fundamental difficulty in expressing this algorithm lies in the one-dimensional data representation in both SDF and SFG semantics. The semantics force the two-dimensional data into a stream. The scanline information naturally described in a two-dimensional structure is lost. To solve this problem, we need a semantics that deal with multidimensional data.

An extension of SDF called MDSDF is described in [7]. Using this semantics, we design a subsampling algorithm using MDSDF in Figure 11. The most important component is the two multidimensional mixer that performs the two types of discard. The 2-tuple labeled at an arc input or output is the data size that the star consumes or produces at each firing of the star.

Figure 11 MDSDF implementation



The first Mixer takes two scanlines and discard the second scanline. The second Mixer takes two vertical strip and discard the even strip. Comparing with Figure 9, This description is much more intuitive and clear compare.

6.2 Edge Detection

The most straightforward way to identify an edge in an image is to find the intensity changes of neighboring pixels. This type of approach is called the gradient base edge detection. In out implementation, we only consider the intensity changes of three directions instead of eight. The three directions that we considered are horizontal to the left, vertical to the top, and diagonal to the north west of the pixel of interest. Our two dimensional filter is as followed:

$$\begin{matrix} -1 & -1 \\ -1 & 1 \end{matrix}$$

The lower right element is the gain of the current pixel. The lower left is the gain of previous pixel, an so on.

The image size in this example is 256 x 256.

6.2.1 approach

To perform filtering, for each pixel, we need to access pixels at previous scanline. Similar to the first example, we have two memory operations: One write and one read that are linked to the same memory space which has size equal to two scanline (512). Refer to Figure 12 for the design in SDF.

There are also two counters. The primary counter increment with modulo-256. The secondary counter offset the output of the primary counter by 128 and then modulo-256. this counter will point to the second scanline in the memory space when the value from the primary counter points to the first scanline and visa versa. The result of the read is the value at the previous scanline.

The value that we write into the memory space is the sum of the intensity of a pixel and its previous pixel. And the value at the read output is the sum of intensity of the two pixels at the same relative position in the previous scanline.

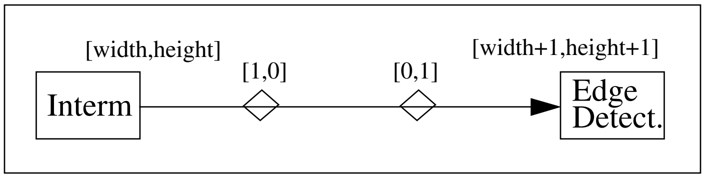
For each pixel, we subtract the pixel value of the previous pixel and then subtract the result of the read. This is the result of our two dimensional filter. Finally, we apply a threshold and sent the result to the output.

CGVSPTarget generates the equivalent graph in VSP software as shown in Fig. 13.

6.2.2 Discussion

The boundary condition is ignored in the algorithm. It would be a complicated task to switch the data path when the algorithm encounters the boundary of a scanline. This again is the result of representing a two dimensional data into a one-dimensional data model. We can easily express the boundary condition in the MDSDF semantics by adding delay in different data dimension. Figure 14 show the delays.

Figure 14. MDSDF implementation



The first delay is along the horizontal direction before the first scanline. The result of this delay is adding a scanline before the first scanline. The second delay is along the vertical direction which means an initial data is added at the beginning of each scanline. The image with the initial condition is now with size:

$$(width + 1) \times (height + 1)$$

This extended image is send to the filtering block for edge detection with initial boundary conditions.

7.0 Conclusion

In this report, the VSP package and the Ptolemy system are briefly introduced. Section 4 explains how to relate the *cycletime* in the SFG semantic to the *repetition rate* in SDF. This project implemented a library of stars that correspond to the operations and operands in SFG. This project also implemented the vsp domain with two targets: **CGVSPTarget** and **VSPTarget**. **CGVSPTarget** generates the corresponding VSP software representation of an universe by reproducing the topology and calculating *cycletime* from *repetition rate*. **VSPTarget** simulates the VSP algorithm in the Ptolemy environment by extending the SDF semantic to support share memory structure.

8.0 Future Work

From our previous discussion, expressing a two-dimensional data with one-dimensional structure is limited. The natural divisions between scanlines are ignored, and this add complexity to our algorithm design. The next step is to consider building a library of MDSDF stars that links to VSP. The VSP domain should base on MDSDF semantics, which represents image algorithm intuitively. In addition an interface between the domain and other computational model in Ptolemy will enable interactions of vsp algorithms with other semantics.

9.0 Acknowledgment

The work that lead to this report would not be possible without the help from my advisor, Edward Lee. The discussions regarding the relationship between cycletime and repetition rate were especially helpful. I would also like to thank Kees Vissers, Joseph T. Buck, Wen-lung Chen, Kennard White, Wei-Yi W. Li, and Louie Yun for their help when I was struggling with the code and the write up.

10.0 References

[1] VSP Support Tools User Guide,(1991) Silicon & Software Systems, Ireland.

[2] G. Essink, E. Aarts, R. van Dongen, P. van Gerwen, J. Korst, K. Vissers, *"Scheduling in Programmable Video Signal Processors"*. Philips Research, Endhoven, The Netherlands.

[3] E.A. Lee, D.G. Messerschmitt, "*Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing"*, IEEE Transactions on Computers, January 1987.

[4] Manual for Ptolemy, University of California at Berkeley, 1992

[5] J. Lim, Two Dimensional Signal and Image Processing, Prentice Hall, Englewood Cliffs,New Jersey, 1990

[6] J.T. Buck, "The Ptolemy Kernel", Technical Report, Memorandum No. UCB/ERL M93/8, University of California at Berkeley, January 1993.

[7] E.A. Lee, "Data Parallelism in Graphical Signal Flow Representations of Algorithms".