



Primitive Programming Manual

June 25, 2008

Address: Blue Horizon Development Software, Inc.
18000 Horizon Way, Suite 200
Mt Laurel, NJ 08054
Telephone: (856) 231-4458
FAX: (856) 231-1403
Internet: www.gedae.com

Table of Contents

1	Introduction.....	5
2	Creating a Primitive	6
	A Simple Primitive	7
	Inplace Streams	9
	E_functions	10
	Parameter Inputs	11
	Data Types	12
3	Token Types.....	13
	Vectors	13
	Matrices.....	15
	Variable Vectors and Matrices.....	16
	Tiled Matrices	17
4	Static Data Flow Parameters.....	20
	Produce and Consume Amounts.....	20
	Local Variables and Reset Methods	22
	Overlap and Hold.....	24
	Delay.....	27
	Iterate	29
5	Families.....	31
6	Dynamic and Nondeterministic	35
	Dynamic Queues.....	35
	Nondeterministic Queues.....	36
7	Summary of Positional and Named Parameter Notation	41
8	Pointer Streams	43
9	Inplace Streams.....	46
	Fixed Inplace Streams.....	46
	Optional Inplace Streams.....	46
	Inplace Output Pointer Streams	47
	Inplace Input Pointer Streams.....	51
	Inplace Tiled Pointer Streams.....	52
	Setting of Tiled Stream Dimensions	53
10	Unmapped Memory	55
11	Persistent Memory	57
12	Segmented Data Flow.....	58
	Segmented Outputs	58
	Segmented Parameters.....	58
	Exclusive Outputs	58
	External State.....	58
13	Runlength Encoded Streams.....	59
14	Cyclic Boxes	61
15	Eval and Trigger Boxes	67
	Eval Boxes	68
	Trigger Boxes.....	69
	GUI Trigger Boxes	70

16	Typedef Boxes	73
	Appendix – Suggested Style Guide	75
	Naming Primitives	75
	Naming Variables	75
	Comments	75
	Index	76

1 Introduction

Gedae applications are developed as graphical hierarchies of boxes. Primitive boxes are the fundamental box types from which flow graphs are constructed. As the basic algorithmic units used to construct applications, they transform data, communicate with external programs, switch data to different parts of applications, display data, and handle I/O. Primitives provide a modular way to introduce code into applications.

Thousands of primitives are supplied by the Gedae core library. They handle data of type `float`, `int`, and `complex` and handle **tokens** that are scalars, vectors, matrices, variable vectors, and variable matrices. Many applications can be built using only the core library primitives. In addition, you have the choice of introducing your own code by writing custom primitives. Some of the reasons for writing custom primitives are:

- To execute a complex algorithm that is not easily decomposed into the available core library primitives
- To process data structure types not handled by the core library
- To execute heritage code that you may later decompose into standard Gedae primitives
- To access a hardware device unique to your configuration
- To encapsulate a custom Graphical User Interface (GUI)
- To communicate with programs running externally to Gedae

The Gedae primitive language is flexible enough to capture a wide range of behavior. Using the language, you can write boxes that do static and dynamic data flow, control flow, state transitions, and parameter transformations. Primitives can be written to produce/consume data from their input/output ports at different rates. The rates can be statically fixed, cyclically changing, or completely dynamic. Primitives can have a parameterized number of inputs and outputs. You can declare state variables that provide persistent storage for the primitive as it is executing. You can also create methods to initialize, reset, save, and restore a primitive's state.

This manual describes how to create custom Gedae primitives and make use of the many features of the primitive language to capture your algorithm's behavior.

2 Creating a Primitive

A Gedae **box** takes input data, performs a calculation, and writes output data. A box can be a **subgraph** or it can be a **primitive**, that is, C-code organized into **methods** that define the computations of the box.

To learn how to write primitives, we begin by creating a new box.

To create a new box, select `Edit`→`Add Box` from the editor, and then type in the filename of the new box in the section labeled `Selection`, as shown in Figure 1. If a box with that name does not exist, then Gedae will ask you whether you wish to create a new primitive or flow graph. For this example we will select `Create Primitive`.

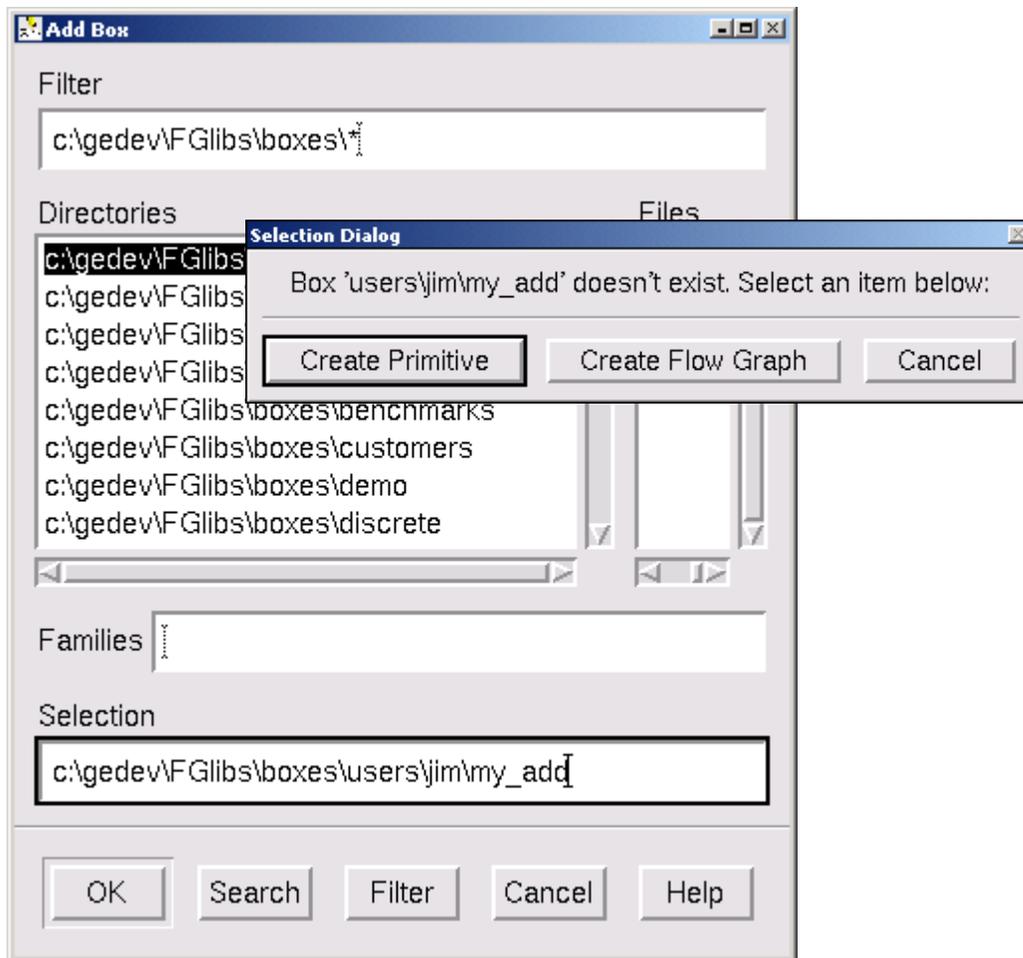


Figure 1 – Creating a new primitive

A Simple Primitive

When `Create Primitive` is selected, an editor similar to the one in Figure 2 pops up. The default content of the editor shows several fields: `Name`, `Type`, `Comment`, `Input`, `Output`, and `Apply`.

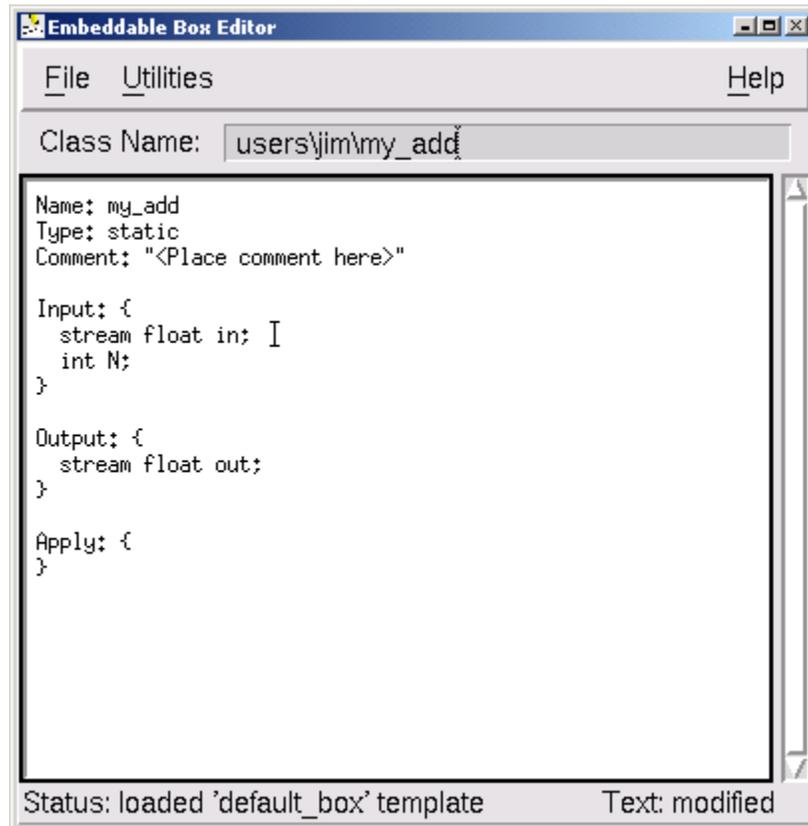


Figure 2 – The Primitive Editor

The `Name` field is the same as the filename of the box, the `Type` field specifies the kind of box, and the `Comment` field is a text string that the box writer can set to describe the functionality of the box. (This `Comment` field is also incorporated into the Gedae flow graph editor and is shown when you press the `Help` button in the `Add Box` or `Search Dialog`.) For this chapter we will only consider boxes of `Type: static`. A simple definition of a box of static type is that the box has a **stream** input and/or output.

The next two fields shown in Figure 2 describe the inputs and outputs of the box. For our add box, we will simply be adding two streams called `a` and `b` and placing the result on an output stream called `out`. To define a stream input of a primitive, declare the input as a `stream` in the `Input` section of the box. For our add box, we declare:

```
Input: {
  stream float a;
  stream float b;
}
```

To define a stream output of a primitive, declare the output as a `stream` in the `Output` section of the box. For our `add` box, we declare:

```
Output: {
  stream float out;
}
```

Now that we have created a new box and have defined its inputs and outputs, we must write the code that performs the calculation. This calculation is done in the `Apply` field (also known as the `Apply` method of the box). The code used in this method is straight C-code, which is extended by built-in variables and functions.

The first and perhaps most important built-in variable is `granularity`. Each input and output stream is implemented as a queue, and the **granularity** of a box defines the number of tokens available on these queues when the `Apply` method is called. Thus, the granularity of a primitive is the number of times the primitive executes its basic algorithm. The box writer must construct the `Apply` method to perform the correct computations for any value of granularity.

The most straightforward way to accommodate different granularities is to construct a **granularity loop** – a for-loop that iterates from 0 to `granularity-1`. For our `add` box, we will construct a granularity loop as follows:

```
Apply: {
  int g;
  for (g=0; g<granularity; g++) {
    out[g] = a[g] + b[g];
  }
}
```

The inputs and outputs we declared above are queues (or arrays) in the `Apply` method. Each array has the same `granularity` number of items when the `Apply` method is called. The for-loop loops over the arrays, adding `a` and `b`, and putting the result in `out`.

After writing the `Apply` method, our `add` box is now complete and should appear in the primitive editor as in Figure 3. We can save the primitive by selecting `File`→`Save` and closing the editor. Now let's construct a graph to test our new primitive. Create a simple graph with two sources (look in `embeddable/stream/source/`) and a scope, which will display the output values (`stream/sink/scope1`). Our `add` box should add the two input streams.

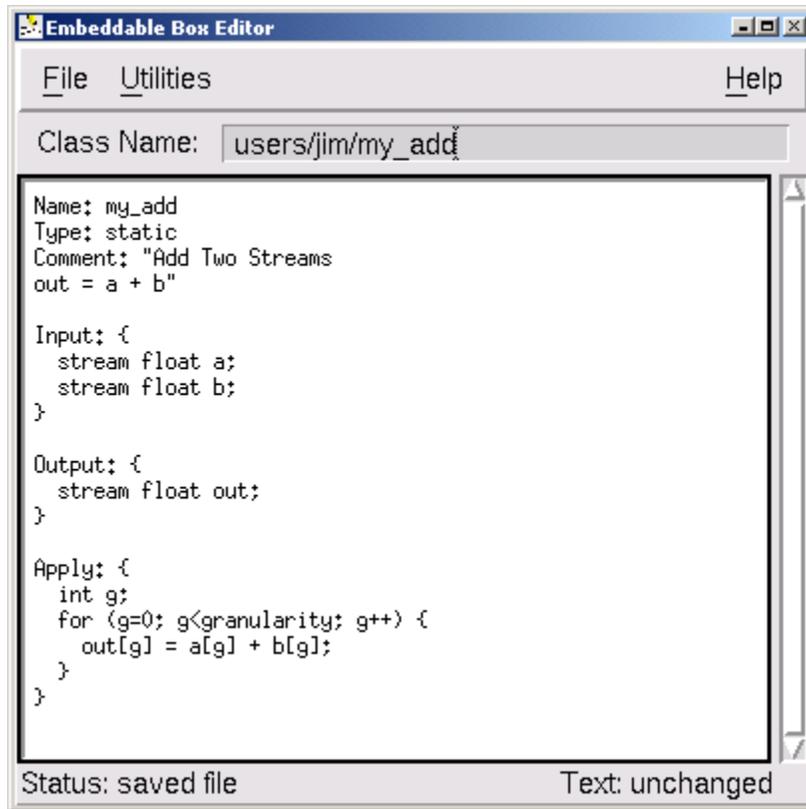


Figure 3 – Final add primitive

Inplace Streams

One improvement we could make to our primitive in Figure 3 is to have the input queue `a` and the output queue `out` use the same address space. Gedae allows primitives to share input and output address space via the word `inplace` in the Output field of the box. For our add box, we can change the Output section to:

```

Output: {
  inplace stream float out=a;
}

```

The stream `out` is declared as an `inplace` stream. We specify that it will take the same place in memory as input `a` by stating `out=a` at the end of the declaration. By making this change to the Output declaration, the Apply method can no longer refer to the stream `out`; only stream `a` exists. Thus, we should change our Apply method to:

```

Apply: {
    int g;
    for (g=0; g<granularity; g++) {
        a[g] += b[g];
    }
}

```

E_functions

Applications created in Gedae run at near hand-coded efficiency. One reason Gedae can make this claim is that the optimized routines supplied by the vendor can be automatically incorporated into the box through the use of the **E_functions**. The E_functions are a set of common functions, including such functions as addition, copying, the FIR filter, and the Fast Fourier Transform. If a box calls an E_function when it executes on an embedded platform, then a version of the routine optimized for the platform is automatically used in the box (if available). The optimized routine replaces the vanilla C-code version of the function that is used when the box executes on the workstation. A full list of the E_functions is available in Appendix C of the Gedae Reference Manual.

To complete the construction of our first box, we consult Appendix C and find that there is an E_function available for adding two floating-point streams:

```

void e_vadd(float *a,int ia,float *b,int ib,
            float *c,int ic,int n);

```

As described in the Appendix listing for this function, a, b, and c are inputs and outputs, the integers ia, ib, and ic are the strides used in looping over the arrays, and the integer n is the number of elements to add. Thus, in order to have an efficient box, we rewrite our Apply method to call the above function:

```

Apply: {
    e_vadd(a,1,b,1,a,1,granularity);
}

```

However, we must also introduce a new field to the primitive in order to run a graph using this new version of the box; it's called the Include field. Each E_function has a corresponding include file, and any include file (as well as, local functions and macros) can be added to a primitive in the Include section of the box. For our box, we add the following before the Apply method and after the Output declaration:

```

Include: {
#include <e_vadd.h>
}

```

Any needed include file can be placed in the `Include` section. For example, if we wished to call `printf` in the `Apply` method, we could add `#include <stdio.h>` to the `Include` section.

Parameter Inputs

The add box we have constructed above adds two streams together. **Parameter** inputs can also be used in boxes. If our add box were to add a floating point parameter to a stream, then we would modify the `Input` section of the box to:

```
Input: {
    stream float in;
    float K;
}
```

Here we have declared a stream input `in` and a parameter `K`. The `Apply` method to add the stream and parameter is:

```
Apply: {
    int g;
    for (g=0; g<granularity; g++) {
        out[g] = in[g] + K;
    }
}
```

Of course, much like our first addition box, we can use an in-place stream to save memory usage and an `E_function` (in this case, `e_vsadd`) to improve the memory usage and efficiency of the routine.

It is useful to provide default values for many parameter inputs, and Gedae provides a way for doing this through the `Init` method. Values set in the `Init` method are used unless the input is connected to a data element on the graph, or the user sets the values using the Gedae Parameter Table. The `Init` method should not perform any calculations other than setting default values. For example, to set the default value of `K` to 1 in the addition box, insert the following `Init` method between the `Include` and `Reset` sections:

```
Init: {
    K = 1.0;
}
```

Data Types

Gedae's core library supports three main data types: `float`, `int`, and `complex`. The `float` type is the standard C single precision floating point number. The `int` type is the standard C integer. The `complex` type is a pair of floats stored side-by-side in memory.

To convert the `my_add` box in Figure 3 to a box that operates on the `int` data type, simply switch the declarations of type `float` in the Input and Output sections of the box to declarations of type `int`.

To convert the `my_add` box in Figure 3 to a box that operates on the `complex` data type, first switch the `float` declarations to `complex` declarations. Then alter the Apply method to:

```
Apply: {
    int g;
    for (g=0; g<granularity; g++) {
        out[g].re = a[g].re + b[g].re; /* add real parts */
        out[g].im = a[g].im + b[g].im; /* add imag parts */
    }
}
```

While Gedae's core library supports `float`, `int`, and `complex` types, a box can be written using any C data type including custom structures. For example, if we wanted to make a stream of 3-D coordinates, we could define and use a structure as in the following example:

```
Input: {
    stream coord3d in;
}
Output: {
    stream coord3d out;
}
Include: {
    struct _coord3d {
        float x;
        float y;
        float z;
    }
    typedef struct _coord3d coord3d;
}
```

Custom primitives must be created to support this new 3-D coordinate type.

3 Token Types

The examples we have investigated thus far have operated on scalars only. Boxes have added scalar streams and scalar parameters. Tokens and parameters can also be arrays.

Vectors

Vector and matrix streams are declared the same way as scalar streams; they also have array dimensions. For example, if we were to redo our add box to add two vector streams, our Input section would be:

```
Input: {
    stream float a[N];
    stream float b[N];
}
```

and the Output section, retaining the property that out and a share the same memory (that is, the queue is inplace), would be:

```
Output: {
    inplace stream float out[N]=a;
}
```

When the Apply method is called, there are granularity **tokens** on each queue, and each token has N **elements**, as declared above. Thus, the Apply method for the vector version of the add box would be:

```
Apply: {
    int g, i;
    for (g=0; g<granularity; g++) {
        for (i=0; i<N; i++) {
            a[i] += b[i];
        }
        a += N;
        b += N;
    }
}
```

Like the scalar version discussed in Chapter 2, this Apply method loops over the granularity; however, inside the granularity loop we must perform the addition for each element of the vector – it is not only a scalar addition. After each vector is processed, the a and b pointers are incremented to point to the next tokens in the streams.

As in Chapter 2, the `E_function` can and should be used, making the box more efficient when run on an embedded processor. This `Apply` method can be rewritten as a call to `e_vadd` with the length input being `N*granularity` as shown in Figure 4.

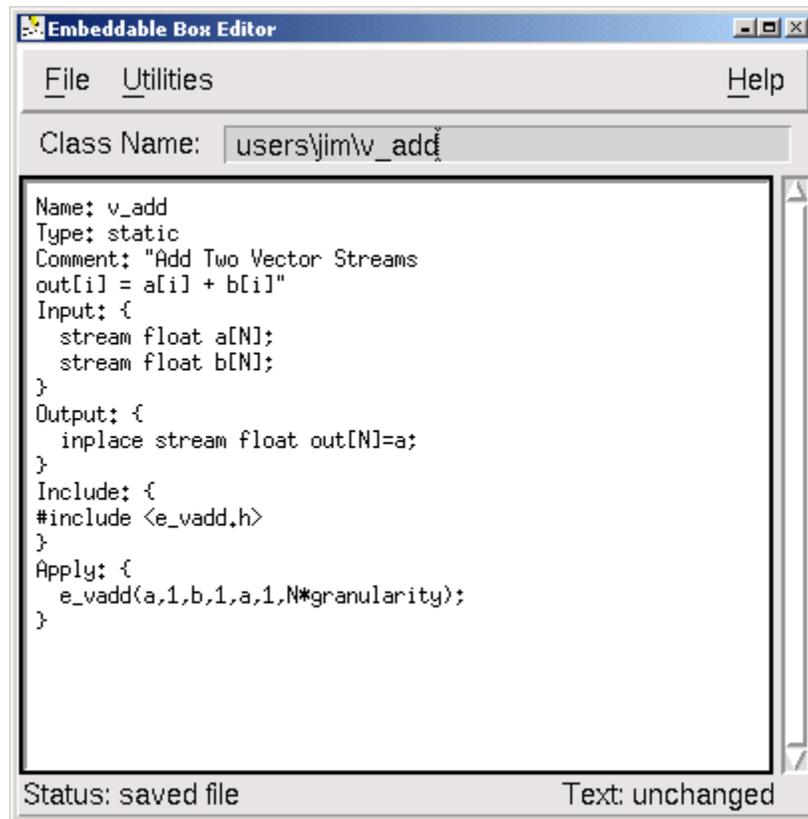


Figure 4 – Final implementation of a vector add

Note that Gedae provides a built-in function `size` that returns the number of values on a queue (whether it be an input or output). In this case, `size(a)` is equal to `N*granularity`. In general, `size(a)` is equal to the number of scalar values in stream `a`.

Now that we have finished implementing this `v_add` box, it is evident how input vector data is passed to the box. However, where does the value of `N` come from? There is no input parameter to the `v_add` box that defines this length. Instead, the value is inherited from upstream boxes; at some point upstream there is a parameter that defines the size of the vector and each downstream box uses that value to define its inputs and outputs. We see in the `v_add` box how the size continues downstream: inputs `a` and `b` know their size `N` from the upstream boxes' definitions, and thus, the box is able to specify the output `out` as having that same token size `N`.

The vector size may be altered as it is passed downstream; we are not limited to merely propagating the value. For example, if a box concatenates two vectors, then it has an Input section:

```
Input: {
  stream float a[N];
  stream float b[M];
}
```

and an Output section:

```
Output: {
  stream float out[N+M];
}
```

(See `embeddable/vector/v_concat` for the full implementation.)

Matrices

Matrices are implemented in a similar manner as vectors. The **matrix** is declared with a row size (number of rows) and a column size (number of columns) like the following example:

```
stream float in[R][C];
```

However, the queue is still implemented like a one-dimensional array and is accessed as such in the `Apply` method of the box. Thus, if we are adding two matrices of size $R \times C$, our `Apply` method can be written as:

```
Apply: {
  int g, i;
  for (g=0; g<granularity; g++) {
    for (i=0; i<R*C; i++) {
      a[i] += b[i];
    }
    a += R*C;
    b += R*C;
  }
}
```

This `Apply` method is a copy of the one written for the previous vector box. We are using a token size of $R \times C$ instead of N . The matrices are stored in row order, that is, element `[i][j]` of an $R \times C$ matrix is located at index $i \times C + j$ in the one-dimensional array.

Variable Vectors and Matrices

When vectors and matrices, as described in the previous two sections, are used, the size of the arrays must be constant during the entire execution of the graph. If the application requires that the size of the arrays change during execution, then variable sized arrays can be used. A stream of variable-sized vectors (or **variable vectors**) is essentially the combination of a fixed length vector stream along with a stream of integers that defines the actual size of the vector. The fixed size defines the maximal size of a vector, but the length stream defines how many elements are actually used in each token. Such a stream is declared as:

```
stream float a[n,Max];
```

In the example above, each token on the stream is allotted `Max` elements. Token `i` on the stream actually uses `n[i]` elements.

When adding two variable vector streams, we use these length streams to determine with which elements to perform the addition, and we use the value of the maximal size to determine where the next token in the stream begins. The primitive that performs this addition is shown in Figure 5.

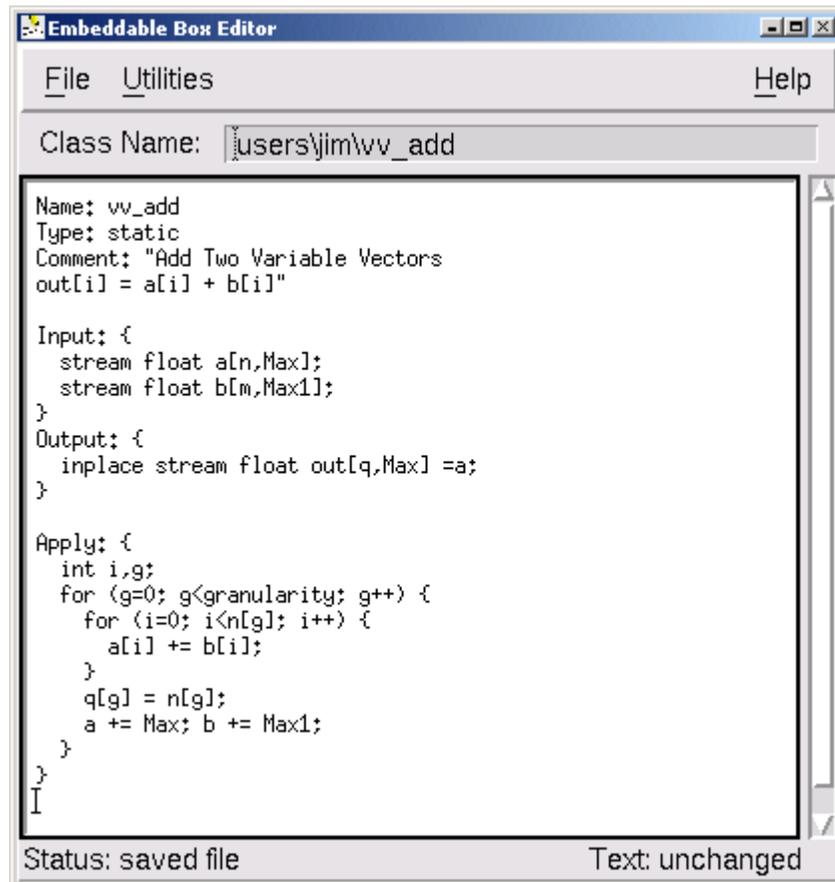


Figure 5 – Variable vector addition

As in previous sections, the primitive loops over the granularity, and at each step must add all the elements of the tokens. This time, however, the number of elements in the token to add is not constant but rather $n[g]$, a value that can be different for each vector. (Note: Even though streams `a` and `out` are in-place, their length streams are not, and the length stream `n` must be copied into the length stream `q`.) To increment the pointers to the next token in the stream, we must increment the pointers by the maximal size.

Variable matrices extend variable vectors in the same way that matrices extend vectors. The queue is still implemented as a one-dimensional array; however, now there are two maximal sizes and two length streams to define the token size. A variable matrix stream is declared as:

```
stream float a[r,MaxR][c,MaxC];
```

In this example, each token on the stream is allotted $MaxR * MaxC$ elements. Token `i` on the queue actually uses $r[i] * c[i]$ elements.

These elements of a variable matrix token are stored in a tightly packed fashion. In other words, all $r[i] * c[i]$ elements are stored at the beginning of the token; the unused elements are all at the end of the token. This layout is shown in Figure 6. Elements in the 4x3 matrix are marked by their row number, and unused elements are colored black.

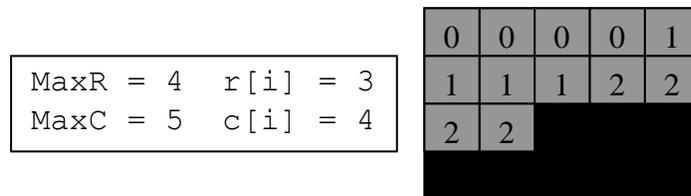


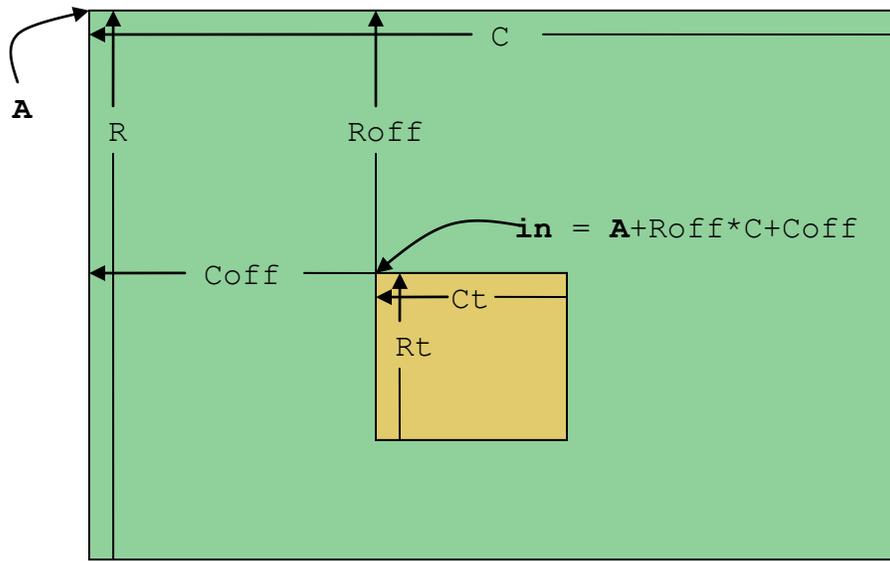
Figure 6 – Layout of a variable matrix

Tiled Matrices

It is sometimes desirable to specify that the input token is a matrix embedded in a larger matrix. This can be done using the tiled dimension notation. For example

```
stream float in[Rt:R][Ct:C]
```

indicates that the input matrix to be processed is of size $R_t * C_t$ but is embedded in a larger matrix of size $R * C$. The pointer `in` is a pointer to the beginning of the matrix within the larger matrix. Below the tiled matrix `in` is seen embedded in a larger matrix `A`. The address of `in` is offset from `A` by a row offset of `RowOff` and a column offset of `ColOff` to give a total address offset of $RowOff * C + ColOff$.



Pointer in points to subtile embedded in larger matrix A

An Apply method that accesses all the input values would be:

```
Apply: {
  int g,r,c;
  for (g = 0; g<granularity; g++) {
    for (r = 0; r<Rt; r++) {
      for (c = 0; c<Ct; c++) {
        float x = in[r*C+c];
        ...
      }
    }
    in += R*C;
  }
}
```

Note that while $R_t * C_t$ values are accessed we index into the matrix with the expression $r * C + c$ instead of $r * C_t + c$ (as we would if the input was declared $\text{in}[R_t][C_t]$). Also we advance to the next token by adding $R * C$ instead of $R_t * C_t$.

Tiled streams can be of any dimensionality – including one-dimensional. In all cases we call the stream tiled – that is we can have tiled vector streams, tiled matrix streams and tiled 3d matrix streams. For example all of the following are tiled streams:

```
stream float a[Nt:N];
stream float b[Rt:R][Ct:C];
stream float c[Xt:X][Yt:Y][Zt:Z];
```

A rule has been established and is enforced by the Gedae parser that if one input or output of a primitive is subtiled then all must be. This rule increases the generality of primitives that have tiled dimensions.

Tiled dimensions are often used in conjunction with inplace pointers to allow zero copy specification of matrix partitioning into tiles. Examples of such primitives are found in `embeddable/matrix/mt_rpart`, `embeddable/matrix/mt_rpartN`, `embeddable/matrix/mt_concatN_ro`. A further discussion of tiled streams as they are used with inplace pointers is found in the section on **Inplace Tiled Pointer Streams**.

The setting of tiled dimensions in the context of a larger Gedae graph follows rules that are described in the section **Setting Tiled Stream Dimensions**.

4 Static Data Flow Parameters

The examples we have discussed have **consumed** granularity tokens from each input stream and **produced** granularity token on each output stream. Actual data flow is often more complicated than these examples. In this section we describe different **static** data flow parameters. Static data flow parameters are ones that are predetermined before the primitive fires. This predetermination is opposed to **dynamic** data flow parameters that can be determined by the primitive when it fires.

Produce and Consume Amounts

The number of tokens consumed or produced on a stream for each execution of the granularity loop of the primitive are called the **consume amount** and **produce amount**. These amounts are assumed to equal 1 unless otherwise specified. We will look at non-unity produce and consume amounts as seen through conversion between scalars, vectors, and matrices. If we are converting a stream of scalars into a stream of vectors, then to produce one vector token we need N scalar **elements** (where N is defined by an input parameter). This conversion is illustrated in Figure 7; if the vector size is 3, three scalar tokens are needed to create one vector token. Even though the number of elements on the queue stays the same, we are still consuming more tokens than we are producing.

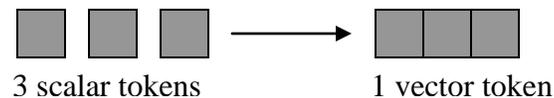


Figure 7 – Scalar to vector conversion with consume amount of 3

The inverse conversion, a stream of vectors to a stream of scalars, will increase the number of tokens on the queue. If the image in Figure 7 is flipped, then we need to create three scalar tokens for each vector token. The granularity of downstream boxes will be affected by this non-unity produce amount.

Let's create boxes that convert between scalars and vectors to see how produce amount and consume amounts are specified in boxes. To convert from scalars to vectors, we need an input parameter to define the size of the output vector and the input consume amount. The scalar to vector conversion box is as follows:

```
Name: s_v
Type: static
Input: {
    stream float in(N);
    int N;
}
```

```
Output: {
  inplace stream float out[N]=in;
}
```

The input parameter N will allow us to specify the size of the vector. Placing this number, surrounded by parentheses, after the name of the input stream specifies a consume amount of N on the stream. Alternatively we can say (consume = N) to specify the consume amount. The output stream is then a vector of length N ; however, it is also an inplace queue that uses the same memory as the input queue. This box does not need an `Apply` method because it is a “no-op” box, that is, it performs no operations, just redefines how data is laid out in memory.

To convert from a stream of vectors to a stream of scalars, we produce multiple tokens on the output for each input consumed as follows:

```
Name: v_s
Type: static
Input: {
  stream float in[N];
}
Output: {
  inplace stream float out(N)=in;
}
```

In the stream-to-vector box, the value in the parentheses on the input defined the consume amount; in the vector-to-stream box, the value in the parentheses on the output defines the produce amount. Alternatively we can specify the produce amount as (produce= N). Each input vector has N elements, and when converted to a stream of scalars, there are N scalar tokens for each vector. Once again, this is a no-op box, as the output elements are the same as the input elements, but we are redefining them as scalar tokens instead of vectors.

When there is non-unity produce or consume amount this causes the primitive to have a non-unity data flow gain. For example, if we investigated the value of the granularity in boxes downstream from a vector-to-scalar conversion, then their granularity would be larger by a factor of N than boxes that performed calculations on the stream of vectors. However, the granularity of the box that performs the conversion is not affected; it only changes the granularity of boxes downstream.

This change of granularity is shown in Figure 8. The `v_ramp` box has a granularity of G , and the conversion from the vector-to-scalar conversion box `v_s` causes the downstream box `sqr` to have a granularity of $N \cdot G$. The granularity of the `v_s` box is G , although, it increases or decreases the granularity for boxes downstream but not for itself. The granularity table (opened from selecting Firing Table... in the Group Control dialog) in Figure 8 shows what happens when a granularity multiplier of 5 is applied. The `sqr` box will handle 80 tokens per execution, 5 multiplied by the vector size of 16. However,

the `v_s` and `v_ramp` boxes will handle 5 tokens per execution; the value of the `granularity` variable inside the box will be 5.

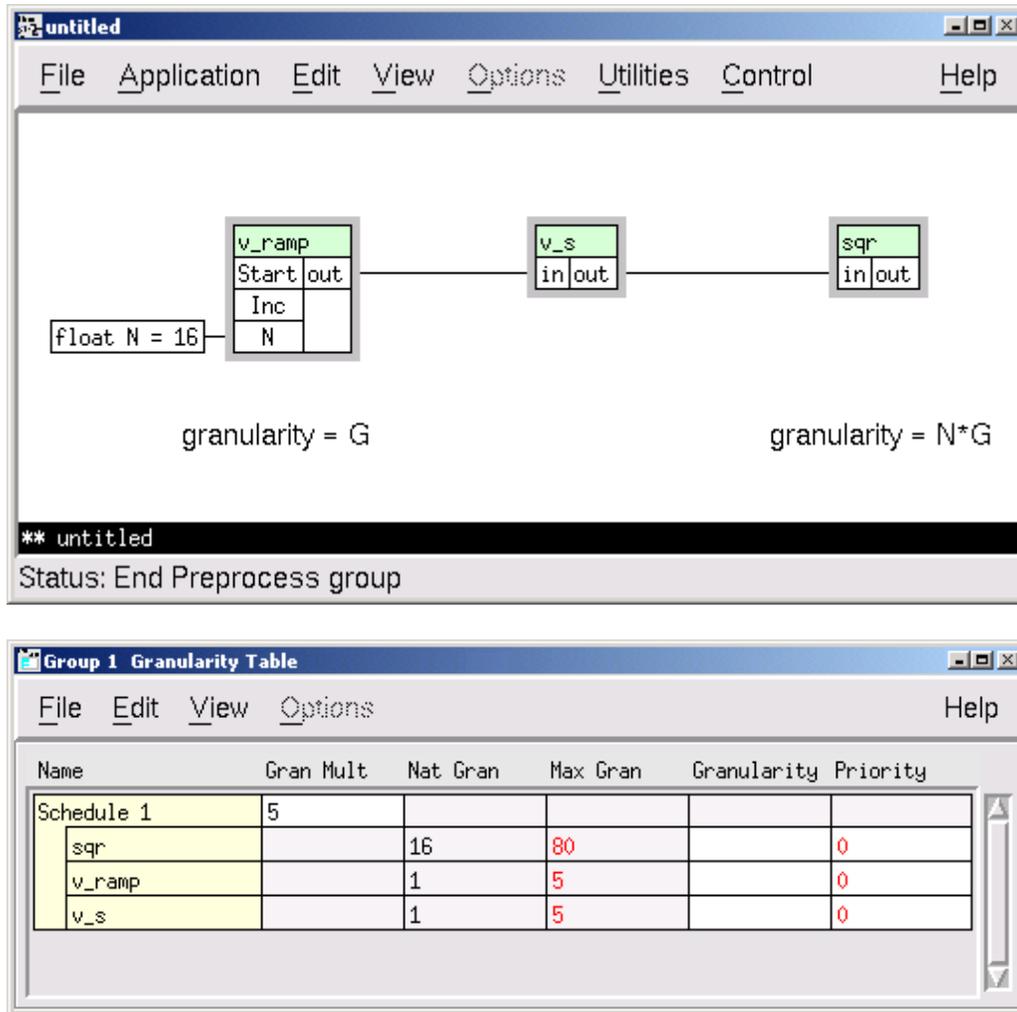


Figure 8 – Change in granularity from a non-unity produce amount

Local Variables and Reset Methods

Sometimes data from previous tokens and previous executions of the `Apply` method is needed in order to perform a computation. One way to store this data is to declare variables in the `Local` section of the box. For example, a box may perform an accumulation of a stream, setting each token on the output as the sum of all tokens that have previously been on the stream. We can calculate this sum by storing the current sum in a local variable, and by setting up a `Local` field as follows:

```
Local: {
```

```
float Sum;
}
```

and then, retrieving and storing this sum in the `Apply` method (the following assumes the output queue and input queue are inplace):

```
Apply: {
    int g;
    in[0] += Sum;
    for (g=1; g<granularity; g++) {
        in[g] += in[g-1];
    }
    Sum = in[granularity-1];
}
```

However, a natural question might be how is `Sum` initialized? This initialization is not done in the `Apply` method and cannot be done there. Instead we must use a different method, a method run only when the graph is reset, that is, the `Reset` method. Like the `Apply` method, the `Reset` method can contain any C-code. In this case, the `Reset` method sets the `Sum` to 0 so that it has a value when the `Apply` method is first called:

```
Reset: {
    Sum = 0;
}
```

The full code of the accumulation box, using the appropriate `E_function`, is shown in Figure 9.

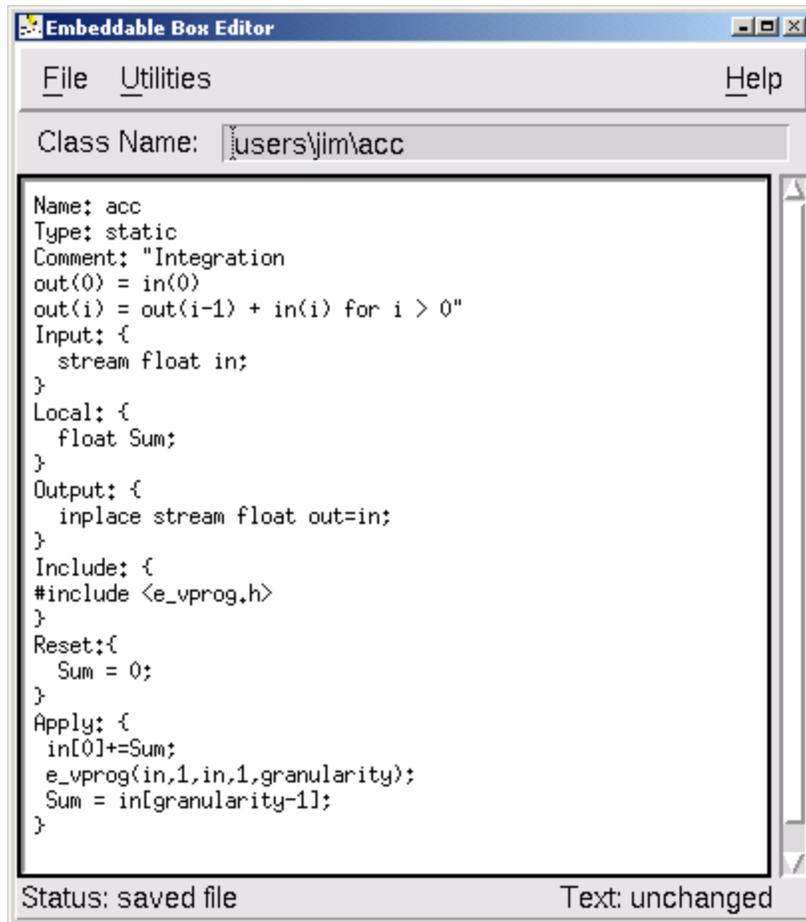


Figure 9 – Implementation of stream accumulation

Arrays can also be declared in the `Local` field of a box, and this method of allocating memory for a local array is more portable in Gedae than using the C routines `malloc` and `calloc` or declaring the vector as a temporary array in the `Apply` method.

Overlap and Hold

Sometimes the previous data and computations needed in the `Apply` method are not one or two values but rather a set of N values. For example, in a sliding window average, the latest N values are averaged for some input parameter N , and that value is placed on the output.

Specifying an **overlap** of $N-1$ on an input stream will retain the last $N-1$ tokens between executions. When a new token is consumed on the stream, the $N-1$ tokens in the overlap can be summed with the new token to produce the result. To declare such an overlap, we add a second number to the parentheses behind an input name:

```

Input: {
    stream float in(1,N-1);
    int N;
}
Output: {
    stream float out;
}

```

The overlap for the stream in is now $N-1$; therefore, in order to specify an overlap, we must also specify the consume amount – in this case, the default value of 1. Each time the Apply method is executed, the first new token on the stream will be stored at `in[N-1]`. The values before `in[N-1]` will be the $N-1$ overlap tokens stored from the previous execution as shown in Figure 10. (These values are 0 if otherwise undefined during the first firings of the box.) In the figure below, the dark gray cells represent old values still on the input queue, and light gray cells represent new tokens on the queue. For each unit of granularity, N items are used in the calculation of the output token.

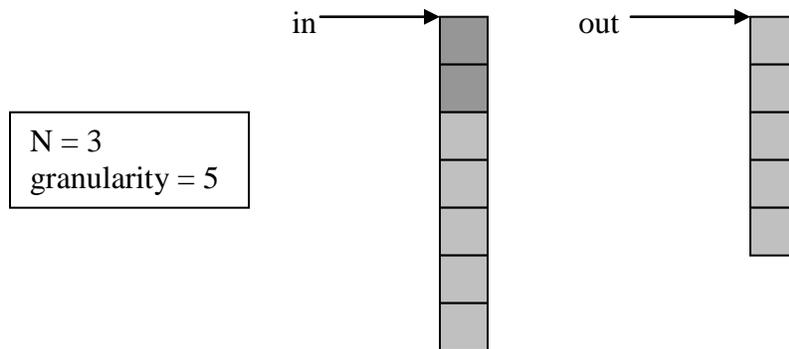


Figure 10 – Data on queues when input has an overlap

Using this overlap, the Apply method can be written as:

```

Apply: {
    int g, i;
    for (g=0; g<granularity; g++) {
        out[g] = 0;
        for (i=0; i<N; i++) {
            out[g] += in[g+i];
        }
        out[g] /= N;
    }
}

```

The innermost loop sums over the length of the window, touching a new token at `in[g+N-1]`, and placing the tally in `out[g]`. (This function is implemented more efficiently in `embeddable/stream/avewin`.)

A **hold** on an output stream performs a similar operation to an overlap on an input stream. When creating an output stream with a hold, you specify that some output tokens be held from being produced until later executions. Let's redo our sliding window average box using an output stream with a hold.

Declaring a hold on an output stream is done much the same way an overlap is specified on an input stream:

```
Input: {
  stream float in;
  int N;
}
Output: {
  stream float out(1,N-1);
}
```

With this declaration, we are holding back $N-1$ tokens on the output. The layout of the data is shown in Figure 11. The light gray tokens represent new values and open space on the queues, while the dark gray tokens represent old values. The output still produces granularity tokens from the top of the queue, and the $N-1$ tokens at the end of the queue are retained for the next execution of the `Apply` method.

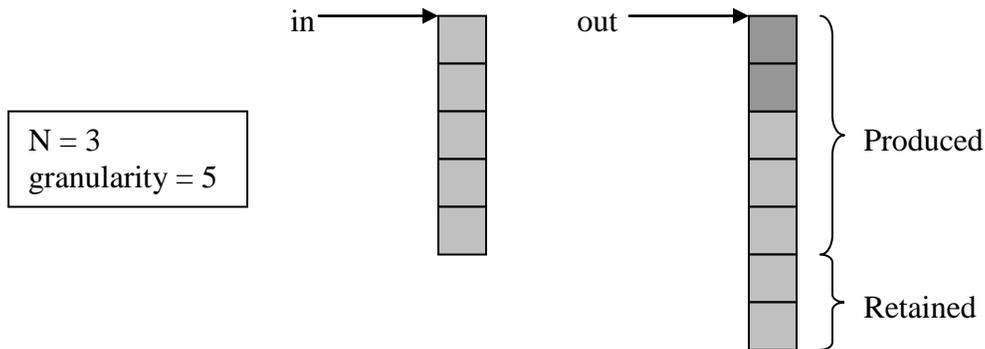


Figure 11 – Data on queues when output has a hold

Thus, the `Apply` method for our sliding window average box can be rewritten for an output stream with hold as follows:

|

```

Apply: {
    int g, i;
    for (g=0; g<granularity; g++) {
        out[g+N-1] = 0;
    }
    for (g=0; g<granularity; g++) {
        for (i=0; i<N; i++) {
            out[g+i] += in[g];
        }
        out[g] /= N;
    }
}

```

In this version of the sliding window average, we are matching input tokens to all appropriate output tokens because during the next execution all the current input tokens will be overwritten and lost. The i -th output token is not produced until N input tokens are used in its calculation.

Delay

An extension of the hold concept is a **delay**. If an output queue is delayed by N tokens, then N zero tokens are inserted at the beginning of the queue. The first token that the primitive can write to is the $N+1$ -th token. To declare a delay, add another number to the numbers in parentheses in the declaration of the output:

```

Input: {
    stream float in;
    int N;
}
Output: {
    stream float out(1,0,N);
}

```

The delay for the stream `out` is N ; thus, to specify this delay we must also specify values for the produced amount and hold of the stream – here they are set to the default values of 1 and 0, respectively. Alternatively we can specify the delay as $(\text{Delay} = N)$. In this case the produce amount and hold take their default values of 1 and 0 respectively.

The layout of the queues is shown in Figure 12. The output stream is delayed by two tokens (shown in white to indicate their value is zero). When the `Apply` method is called for the first time, the `out` pointer points to the first token after the delay, in this case, the third token on the output stream.

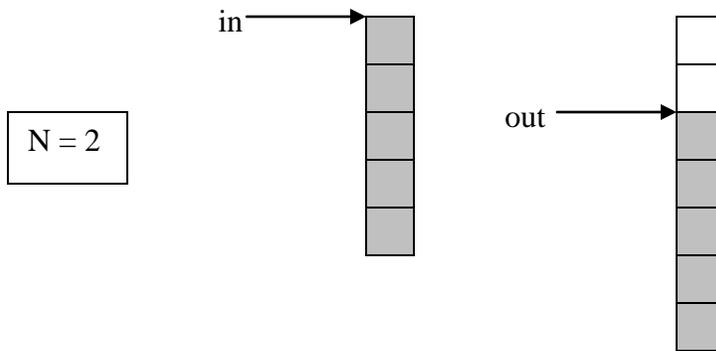


Figure 12 – Delayed output at the first execution of box

Inputs streams can also have a delay, and the effect is similar to what would happen if the output that feeds the input were written with a delay. A delayed input is declared in the same manner as a delayed output:

```

Input: {
  stream float in(1,0,N);
  int N;
}
Output: {
  stream float out;
}

```

With this declaration, the input in has a delay of N. The layout of the queues at the first execution of the box is shown in Figure 13. Zero tokens (colored white) are inserted for the delay before the input tokens from the upstream box (colored gray) are made available on the queue.

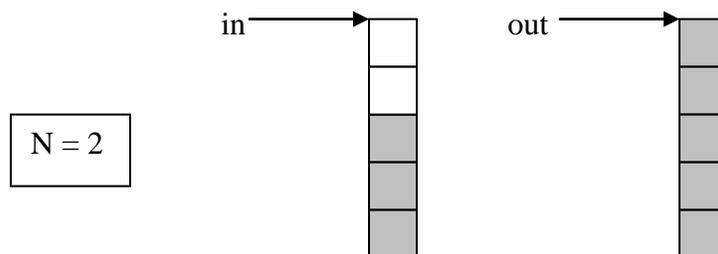


Figure 13 – Delayed input at the first execution of the box

A box with input delay can initialize the delay tokens to values other than zero in its `Reset` method. For example, a box could specify an input delay of N and initialize the initial N values to the values stored in an input parameter `C[N]` as follows:

|

```

Input: {
    stream float in(1,0,N);
    float C[N];
}

Reset: {
    int i;
    for (i=0; i<N; i++) in[i] = C[i];
}

```

Iterate

The iterate parameter specifies how many firings of a primitive are needed to consume and input token or produce an output token. Its meaning is the opposite of the produce or consume amount. For example if the consume amount is set to 3 then every firing of the primitive will consume 3 tokens. If the iterate amount of the input is set to 3 then 3 firings of the primitive are needed to consume 1 token. To declare the iterate amount, add another number to the numbers in parentheses in the declaration of the output: For example:

```

Output: {
    stream float out[R*N][C](1,0,0,N);
}

```

Or use the named parameter list to specify the iterate as

```

Output: {
    stream float out[R*N][C](iterate = N);
}

```

The above specification says that the box must fire N times before it produces one token on the output. The same syntax is used to specify the iterate on an input. Recall that the v_s box was described using the produce amount as:

```

Name: v_s
Type: stream
Input: {
    stream float in[N];
}
Output: {
    inplace stream float out(N)=in;
}

```

This box can be described using an input iterate value as:

```

Name: v_s

```

```

Type: stream

Input: {
    stream float in[N] (1,0,0,N);
}

Local: {
    int which; /* points to the next index in the input
                vector to be output to the scalar output */
}

Output: {
    stream float out;
}

Reset: {
    which = 0;
}

Apply: {
    int g;
    for (g = 0; g<granularity; g++) {
        out[g] = in[which++];
        if (which == N) {
            which = 0;
            in += N;
        }
    }
}

```

Note that the granularity of execution does not need to have any relation to the vector size N . As a result we need to keep track of how many tokens in the input vector have already been copied to the output. This is the purpose of the `which` variable. After each N firings of the granularity loop the `which` variable is reset to 0 and we advance the input to the next token.

Since `iterate` on an input dictates almost the same behavior as `produce` on an output and `iterate` on an output dictates almost the same behavior as `consume` on an input why specify a new data flow parameter? The reason is that the `iterate` parameter used in conjunction with tiled pointer inputs and outputs allows primitives to be written that extracts subtiles out of a larger matrix. An example of this is the

5 Families

Inputs and outputs to primitives can be grouped into **families**. An example is shown in Figure 14; a family of source boxes feeds an input to a box named `sum` that sums all the sources together.

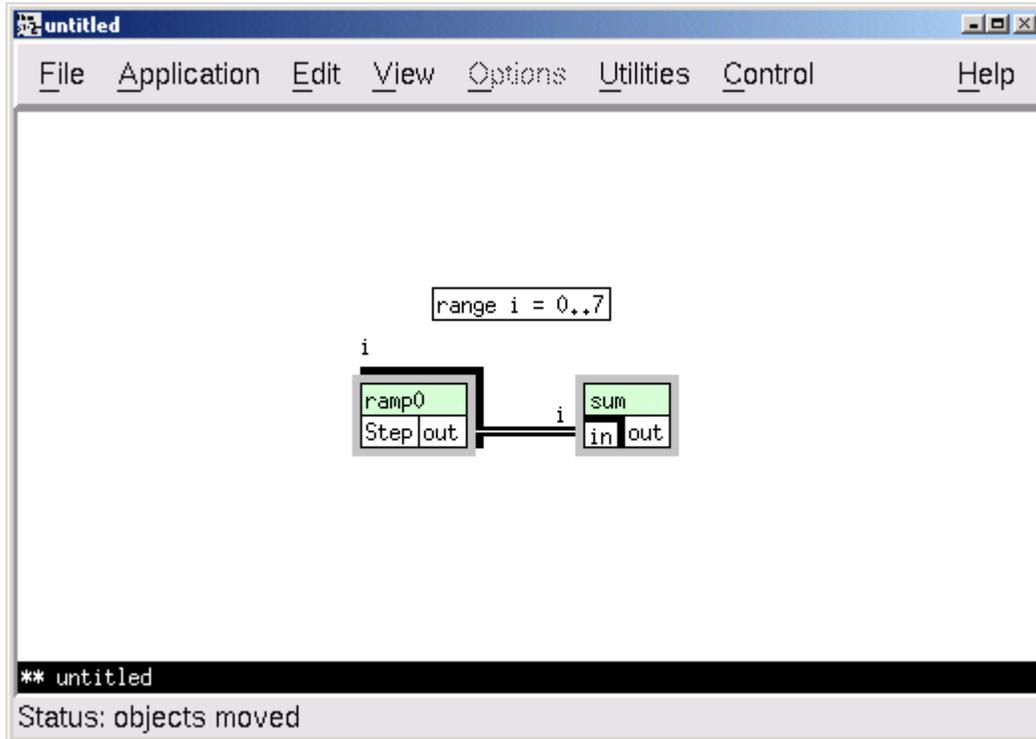


Figure 14 – A family input to a primitive

The family input to the `sum` box is declared with a pre-index as follows:

```
Input: {
    stream float [N]in;
}
```

The value of `N` is inherited from the `range` on the canvas (`N` will be set to eight in Figure 14). In the discussion of token types in Chapter 3, the queue is always stored as a one-dimensional array despite the token type or granularity. Different elements in the vector and different tokens in the stream are accessed through a single pointer. The use of a family introduces a second pointer.

It is useful to understand why families introduce this second pointer. In the graph in Figure 14, each `ramp0` box has an output `out`, and it is these queues that are accessed on the input of `sum`. The family input to the `sum` box is an array of pointers to these

ramp0 output queues; making the data available as a one-dimensional array would require copying large amounts of data.

The layout of these queues is illustrated in Figure 15.

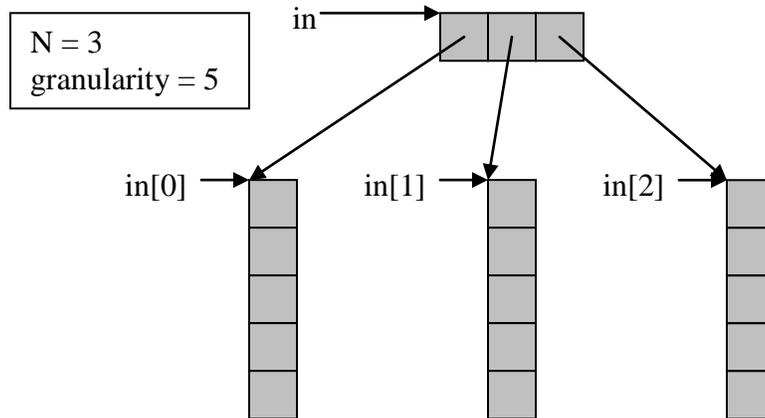


Figure 15 – Layout of a family input

To get the pointer to the f -th family member of input in , we use the index $in[f]$. This family member's value is a pointer to a queue with granularity tokens. To see this, look at the Apply method for the sum box below:

```
Apply: {
  int j, g;
  for (g=0; g<granularity; g++) {
    out[g] = in[0][g];
  }
  for (j=1; j<N; j++) {
    for (g=0; g<granularity; g++) {
      out[g] += in[j][g];
    }
  }
}
```

In this Apply method, we initialize the output queue by copying over the 0-th family member of the input. Then, we add the other family members' queues to the tally. (The Apply method can be written more efficiently using E_functions as it is done in `embeddable/stream/sum`.)

While multiple dimensions of families are possible on the canvas, primitives only allow one-dimensional family inputs and/or outputs. To use a primitive on a multiple-dimensional family, one must first use a route box to collapse the multiple dimensions into one dimension.

In the `sum box` example, all the vectors have the same vector length because they will be added together. However, this restriction on the vector length of family members is not a requirement. If a vector is being partitioned into a family of subvectors and the length is not evenly divided, then the subvectors can be of different lengths. Figure 16 shows such a scenario: a vector of length 10 is being partitioned into 3 parts; since 3 does not evenly divide 10, two of the subvectors have length 3 and the other subvector has length 4.

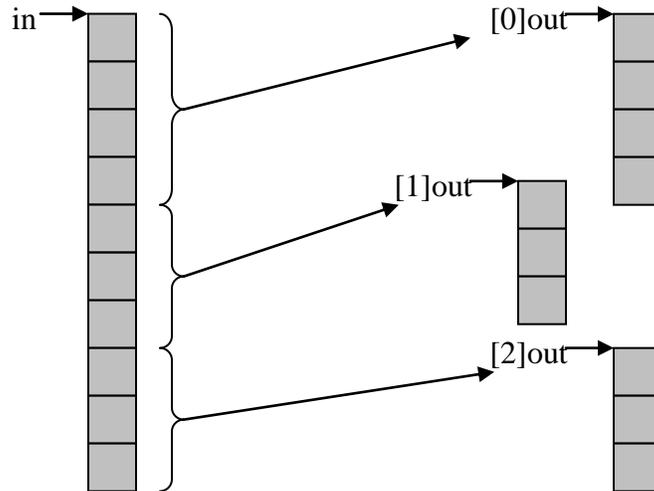


Figure 16 – Partition vector into subvectors of different lengths

To declare this family output of vectors of different lengths, we introduce a range variable to the family index using the colon (`:`) notation, and then use that range variable to define the lengths of the arrays (this index can also be used to compute produce amounts and consume amounts and any other dataflow parameters). For our vector-partitioning example, the output is declared as follows:

```
Input: {
  stream float in[N];
}
Output: {
  stream float [i:M]out[(i < N%M) ? N/M + 1 : N/M];
}
```

The input vectors have a length N and are partitioned into M parts. We use the variable i to cover the range 0 to $M-1$, and the variable i is used to define the size of each array in the family. A conditional is used: if i is less than $N\%M$, the length is $N/M + 1$, otherwise it is N/M . In the example in Figure 16, N is 10 and M is 3; therefore, the first vector gets $N/M+1 = 4$ elements, and the rest of the vectors get $N/M = 3$ elements. The `Apply` method must be written to accommodate these lengths. An efficient way to implement this partitioning is to loop through the first $N\%M$ family members, copying subvectors of length $N/M+1$, and then to loop through the rest of the family members,

copying subvectors of length N/M . (To view the implementation of this partitioning, open `embeddable/vector/v_part`.)

Family members of inputs can also have different array dimensions. To see how this is implemented, inspect the `v_part` counterpart `embeddable/vector/v_nconcat`, which concatenates a family of input vectors. The inputs and outputs of this box are declared as follows:

```
Input: {
  stream float [i:M] in[N[i]];
}
Output: {
  stream float out[sum(N,M)];
}
```

By using the colon notation, `i:M`, on an input family dimension, the array sizes of the family members are defined by a vector `N`, where `N[i]` is the size of family member `i`. Because the output of this concatenation box has a length equal to the sum of all the input lengths, the built-in function `sum` can be used to sum over the family member sizes.

6 Dynamic and Nondeterministic

In every example discussed in the first five chapters of this manual, the number of tokens consumed on input queues and produced on output queues is **static** (known at runtime). Queues and boxes are not always static.

Dynamic Queues

A **dynamic** queue allows you to specify how many tokens are needed in order to execute, but allows you to actually use only some of these tokens. The number of tokens needed to execute is called the **threshold**. For inputs the threshold indicates how many tokens must be available on the queue for the primitive to **fire** once (**execute** at a **granularity** of 1). The primitive can consume up to $\text{granularity} \times \text{threshold}$ number of tokens when it executes. For outputs the threshold indicates how much space must be available on the queue before the primitive can fire once. The primitive can produce any number of tokens up to $\text{granularity} \times \text{threshold}$.

To understand dynamic queues, let's investigate a box that converts variable vector tokens into scalar tokens. This conversion is shown in Figure 17. The variable vector input has space allocated for four elements but has an actual length of three (the black element is empty); three scalar tokens are produced on the output and the unused element is discarded.



Figure 17 – Conversion from variable vector to scalar

To declare a dynamic output queue with a threshold of `Max`, we indicate it is a `dynamic stream`, as follows:

```
Input: {  
    stream float in[n,Max];  
}  
Output: {  
    dynamic stream float out(Max);  
}
```

The output stream `out` is a dynamic stream that will produce a maximum of `Max` tokens for each input token. This queue's threshold is specified in the same way as the produce amount is specified and if not specified the threshold is takes the default value of 1. In this conversion, the length stream `n` is used to determine how many tokens are produced

on the output. The `Apply` method merely copies elements from the variable vectors to tokens on the output stream, looping through the granularity, as follows:

```
Apply: {
    int g;
    int cnt = 0;
    for (g=0; g<granularity; g++) {
        e_vmov(in,1,out+cnt,1,n[g]);
        cnt += n[g];
        in += Max;
    }
    produce(out,cnt);
}
```

The variable `cnt` tallies the number of tokens placed on the output. Because variable vectors are constructed such that `n[g]` is never larger than `Max`, each variable vector will not create more than `Max` scalar tokens on the output. After all the elements are copied over, a call is made to the function `produce`.

The function `void produce(Queue name, int count)` informs Gedae how many output tokens were actually created by the `Apply` method. This function should only be called once at the end of the `Apply` method after all items in the granularity have been computed.

Dynamic inputs can also be used, using the function `void consume(Queue name, int count)` at the end of the `Apply` method to tell Gedae how many input tokens should be consumed.

Nondeterministic Queues

When using **nondeterministic** queues, there is no guarantee that when the `Apply` method is called there will be sufficient tokens on the queues to perform the computation. You must use built-in functions to determine whether enough space is available (if it is an output queue) or if enough new tokens are available (if it is an input queue).

The functions and variables that are used to handle nondeterministic queues are shown in Table 1. There are two options for determining how much space to use on a nondeterministic queue.

The first option is to use the `avail` function to query how much space is available on each nondeterministic queue in the box. This information determines how much space the `amount` function should prepare for use. Using the `avail` function is more flexible but also less desirable in most cases.

The second option is to simply call the `amount` function on each nondeterministic queue specifying how much space is desired on each queue. If the desired amount of space is available, then the `amount` function prepares it for use. If the desired amount is not available, then the value of the variable `queues_ready` is set to 0, and this variable must be used to determine if the computation should be continued. This second option is advantageous because calls to `amount` inform Gedae how many tokens are needed on each input. Thus, if the desired number of tokens is not available the first time, then this information allows Gedae to avoid calling the `Apply` method again until it has all the tokens necessary to successfully execute.

Function Declaration	Description
<code>void produce(Queue name, int cnt);</code>	Indicates at the end of the <code>Apply</code> method that <code>cnt</code> tokens were created on the output queue
<code>void consume(Queue name, int cnt);</code>	Indicates at the end of the <code>Apply</code> method that <code>cnt</code> tokens were used from the input queue
<code>int avail(Queue name);</code>	Determines how many tokens are available on the input or output queue
<code>void amount(Queue name, int cnt);</code>	Prepares <code>cnt</code> tokens for use on the input or output queue
<code>int queues_ready;</code>	The value is 0, if <code>amount</code> has failed during the current execution of the <code>Apply</code> method; 1 otherwise

Table 1 – Functions used with dynamic and nondeterministic queues

To investigate nondeterministic queues, consider a box that merges two streams together. A control stream specifies from which input to take a token. The two streams that are being merged together must be nondeterministic because the order in which the streams are merged is determined at runtime through the control stream. If the queues were static or dynamic, then tokens would have to be available on both inputs before the box would be able to execute. With nondeterministic queues, however, if one input queue is full and the control stream says to take tokens from that input, then the box is able to execute whether or not the other input has tokens.

To declare an input or output as nondeterministic, indicate it is a `nondet stream`. For the merge box, we have two nondeterministic streams and a static control stream:

```
Input: {
    nondet stream float in0;
    nondet stream float in1;
    stream int c;
}
```

The merge of two streams can either be **complete** or **incomplete**. If the merge is complete, then an output token is produced for each token on the control stream (values

on the control stream indicate either “copy from in0” or “copy from in1”). In this case, the output is static, and the number of output tokens is known before runtime. If the merge is incomplete, then there are more control tokens than output tokens (values on the control stream indicate either “copy from in0,” “copy from in1,” or “do not produce a token for this value”). In this case, the output is dynamic. Let’s concentrate on a complete merge, with a static output stream:

```
Output: {  
    stream float out;  
}
```

To perform the merge, the box examines the control stream and counts how many tokens are needed on each of the two nondeterministic inputs. Then, it uses these counts to call the `amount` function for each input. If the `amount` function succeeds, then the variable `queues_ready` will be set to 1, in which case, the box performs the copy from the input streams to the output stream; otherwise, it does nothing and waits for the next execution of the `Apply` method.

The code for the complete merge of two streams is shown in Figure 18. The local variable `c_examined` marks whether the last execution consumed tokens from the input. If the last execution consumed tokens, the `Apply` method examines the `c` input in order to determine how many tokens to take off the inputs. Variables `n0` and `n1` are counters that determine the number of tokens to consume.

After the control stream has been examined, the calls to `amount` inform Gedae how many tokens are needed on each input. If these calls to `amount` are successful, that is, if the number of tokens needed is available, then the test of `queues_ready` will allow the box to copy tokens from the inputs to the output and consume the tokens from the inputs. If `queues_ready` is false, then these tokens are consumed during the next execution of the box.

Note the following procedure used to handle the nondeterministic queues in the complete merge box:

1. Count the number of tokens needed on each nondeterministic queue
2. Call the `amount` function to see if that number of tokens is available
3. Test the `queues_ready` variable to see if the `amount` functions succeeded, and the box is ready to execute
4. If the box is ready to execute, perform the data copies and consume data from the input queues

```

Name: merge_c
Type: static
Comment: "Complete Merge on Stream
if (c==0)
    copy and consume token from in0 to out
else
    copy and consume token from in1 to out"
Input: {
    nondet stream float in0(1);
    nondet stream float in1(1);
    stream int c(1);
}
Local: {
    int c_examined;
    int n0; /* number of control tokens set to 0 */
    int n1; /* number of control tokens set to 1 */
}
Output: {
    stream float out(1);
}
Reset: {
    c_examined = 0;
}
Apply: {
    int i;
    if (!c_examined) {
        /* determine number of control tokens of each type */
        n0 = n1 = 0;
        for (i=0; i<granularity; i++) {
            if (c[i]==0) n0++;
            else          n1++; /* if not a zero assume a 1 */
        }
        c_examined = 1;
    }
    amount(in0,n0); /* declare how much we need on each input */
    amount(in1,n1);
    if (queues_ready) {
        int j0=0;
        int j1=0;
        for (i=0; i<granularity; i++) {
            if (c[i]==0) {
                out[i] = in0[j0++];
            } else {
                out[i] = in1[j1++];
            }
        }
        consume(in0,n0);
        consume(in1,n1);
        c_examined = 0;
    }
}

```

Status: saved file as: users\jim\merge_c Text: unchanged

Figure 18 -- Nondeterministic queues in a complete merge of two inputs

Nondeterministic output queues are handled in a similar way to nondeterministic input queues. The queues must be prepared for use at the beginning of the `Apply` method, and then the `produce` function is called to produce the tokens on the output for downstream boxes.

If nondeterministic queues are grouped into a family, then each family member is treated individually. The `amount` function should be called on each applicable family member, and then `produce` or `consume` should be called on each applicable family member. To see an example of how to use a nondeterministic family, compare the `merge_c` box in Figure 18 to the primitive `embeddable/stream/logic/mergef_c`, which performs a complete merge of a family of inputs. The code for the `mergef_c` is organized in much the same way as the code for the `merge_c`. Below is the procedure used by the `mergef_c`:

1. Examine the control stream and keep counters, using local array `n[]`, on how many tokens are needed on each queue in the family
2. Call the `amount` function on each queue:

```
for (j=0; j<N; j++) {
    amount(in[j],n[j]);
}
```

The `j`-th family member is `in[j]`, and the counter `n[j]` was used in step 1 to count how many tokens are needed on that queue

3. If the `amount` function was successful as tested by the variable `queues_ready`, then copy the data to the output and call `consume` on each family member:

```
for (j=0; j<N; j++) consume(in[j],n[j]);
```

If all of a box's queues are nondeterministic, then the box can be labeled `Type: nondet` (instead of `Type: static`). Labeling each input and output queue as `nondet` is equivalent to changing the `Type` field to `nondet`.

7 Summary of Positional and Named Parameter Notation

Gedae supports both a position dependent notation and a named parameter notation for specifying data flow parameters. In the preceding sections only the positional notation is shown. This notation allows the specification of dataflow parameters by their order in a parenthesized list. The named parameter notation is more verbose but easier to read as the user does not need to memorize the position of the parameters to interpret the input or output data flow specification. For example static outputs have data flow parameters produce, overlap, delay and iterate. These parameters can be specified as

```
stream float out(produce = 1, delay = 0, overlap = 5,
iterate =1);
```

Since the default values of produce = 1, delay = 0, overlap = 0 and iterate = 1 only the parameters that differ from the default need to be specified. So the above can be shortened to

```
stream float in(overlap = 5);
```

In the position dependent notation the output parameters are given in the order (produce,overlap,delay,iterate). In this notation the output above could be declared as

```
stream float out(1,5,0,1);
```

Since the iterate and delay assume their default values they do not need not be specified. This shortens the declaration to

```
stream float out(1,5).
```

In the position dependent notation we still need to specify the produce amount of 1 even though it takes its default value – so that the overlap value will appear in position 2.

Inputs are handled similarly to outputs. All the data flow parameters are named as the input with the exception of produce which for inputs is changed to consume. Thus an input declaration might be

```
stream float in(consume = D, overlap = D-1, interate = N);
```

This would correspond to a position dependent declaration

```
stream float in(D,D-1,0,N);
```

Dynamic streams allow a single data flow parameter, threshold, to be set and nondeterministic queues allow a single parameter capacity to be set. For dynamic queues a threshold of $N+1$ can be set either as

```
dynamic stream float out(N+1);
```

Or

```
dynamic stream float out(threshold = N+1);
```

For nondet queues a capacity of $C-1$ can be set either as

```
nondet stream float out(C-1);
```

or

```
nondet stream float out(capacity = C-1).
```

8 Pointer Streams

The goal of **pointer** streams is to eliminate unnecessary copying of data. Gedae primitives must fill their output streams and sometimes this filling is just copying the data from one buffer to the output buffer. Boxes that encapsulate an input device are an example of a box that does unnecessary copies. By their nature input devices often produce the highest bandwidth data in the system and extra copies at the input device are the most expensive.

The copying problem results from the fact that Gedae primitives execute out of memory specially allocated for the primitives called **schedule memory**. A primitive will process its inputs directly out of schedule memory and dump its results directly into schedule memory without copying. Thus – within a static schedule – dataflow is implemented without unnecessary copies; however, for an input device the input is often initially written to a buffer available in a buffer area outside of schedule memory. This need for an external buffer is a function of how the input device drivers were written and is outside of the control of Gedae. A double buffering scheme is often used by the input device so that data can be put into one buffer by the device while being processed from another buffer. A Gedae primitive encapsulating the input device would have to copy the data out of the filled input device buffers into Gedae schedule memory.

Instead of copying the data from an external buffer to schedule memory – a primitive with a pointer output sets the output to point to the external buffer. To express this situation a new keyword, `pointer`, can be used to modify a stream output.

For example a stream output can be declared as:

```
pointer stream float out[N];
```

The keyword `pointer` indicates that schedule memory will not be allocated for the destination boxes but instead the memory will be provided directly by the source primitive. To use pointers several problems need to be solved. First since each token is a pointer and need not be contiguous with the next pointer – pointer boxes must fire at a granularity of 1. Gedae detects at pointer boxes at compile time and subschedules them if necessary so that they run at a granularity of 1.

A second problem is that since the pointer may need to be reused at a later time there needs to be a way to signal when the downstream primitives are finished using the pointer. This is handled by registering a release method when the output pointer is set. A new built in Gedae function, `set_ptr`, has been provided that allows the primitive `Apply` method to set the output to the external pointer and register a release function. Function `set_ptr` takes four arguments:

```
set_ptr(<output_stream>, void *src_ptr,  
        void (*release)(void *, void *, int), void *handle)
```

where:

- `<output_stream>` is the name (without quotes) of the output pointer stream
- `src_ptr` is the pointer to which the output stream is to be set
- `release` is the function pointer that will release the `src_ptr` when all primitives using the pointer have completed.
- `handle` is the a handle passed as the second parameter to `release` to provide any information in addition to the pointer needed to release resources

The function `release`, which should be defined in the primitives `Include` section takes three parameters:

```
release(void *src_ptr, void *handle, int n)
```

where:

- `src_ptr` is the second parameter passed to `set_ptr`
- `handle` is the fourth parameter passed to `set_ptr`
- `n` is the number of tokens in the pointer which will be the granularity of the box calling `set_ptr` times the produce amount of the output stream.

The `release` function is called by the primitive `internal/release` which is automatically inserted by the Gedae scheduler. The `release` primitive is scheduled to execute the `release` function after the last primitive using the pointer executes. If no `release` function is required `set_ptr` can either be called with a `release` function of 0 or better called with just the first two parameters. If `set_ptr` is called with just the first two parameters the `internal/release` primitive is not added to the graph.

In addition to the `set_ptr` function for static outputs, a second function – `produce_ptr` – was developed for use with dynamic outputs. The `produce_ptr` function is identical to `set_ptr` except for an additional parameter indicating the number of samples that need to be produced. Function `produce_ptr` is used on outputs of type `pointer stream dynamic` or `pointer stream nondet` and combines the function of `set_ptr` and `produce`. The prototype for `produce_ptr` is:

```
produce_ptr(<output_stream>,void *src_ptr, int tokens  
          void (*release)(void *, void *, int), void *handle)
```

where:

- `<output_stream>` is the name (without quotes) of the output pointer stream
- `src_ptr` is the pointer to which the output stream is to be set

- `tokens` is the number of tokens to be produced
- `release` is the function pointer that will release the `src_ptr` when all primitives using the pointer have completed.
- `handle` is the a handle passed as the second parameter to `release` to provide any information in addition to the pointer needed to release resources

Note if an output pointer stream is to be set to point to input memory it is necessary to use the `inplace` modifier as described in the section on **Inplace Output Pointer Streams**. If the `inplace` modifier is not used Gedae does not know that the input memory is still in use after the primitive fires and may assign other primitive outputs to this memory area. This type of problem causes the primitives using the pointer output data to operate on data that has been corrupted.

9 Inplace Streams

As previously discussed Gedae allows a primitive to share input and output address space via the word `inplace` in the `Output` field of the box. In this section we describe some variations of the `inplace` stream concept.

Fixed Inplace Streams

We previously described the Gedae `add` primitive whose output is declared to be `inplace` with it's a input. For completeness we repeat the discussion here. This primitive can be written as:

```
Name: add
Type: static
Input: {
    stream float a;
    stream float b;
}
Output: {
    inplace stream float out=a;
}
Apply: {
    int g;
    for (g = 0; g<granularity; g++) {
        a[g] = a[g] + b[g];
    }
}
```

Note that since the output `out` is declared to be `inplace` with `a` that `out` does not appear in the `Apply` method. Instead the `Apply` method pointer `a` is the pointer to both the input stream `a` and the output stream `out`.

Optional Inplace Streams

Inplace streams can save memory but are occasionally cause inefficiencies. For example if a primitive output is connected to two `inplace` inputs then an internal/copy primitive must be automatically added by Gedae in front of one of these inputs. The copy eliminates the memory savings provided by the `inplace` stream and causes extra overhead. To avoid this problem streams can be declared to be optionally `inplace` with one or more streams. The notation for this is illustrated in the following example:

```
Name: add
Type: static
```

```

Input: {
    stream float a;
    stream float b;
}
Output: {
    inplace stream float out={a,b};
}
Apply: {
    int g;
    for (g = 0; g<granularity; g++) {
        out[g] = a[g] + b[g];
    }
}

```

In the above example `out` is declared to be optionally inplace with either `a` or `b`. Since we don't know if `out` will be inplace with one or none of these inputs the stream `out` must appear explicitly in the `Apply` method. Gedae will attempt to make `out` inplace with `a` but if a copy box is needed as a result it will then attempt to make `out` inplace with `b`. If a copy box is still needed Gedae will make `out` out-of-place with both `a` and `b`.

Inplace Output Pointer Streams

An output pointer stream can be made inplace with an input stream to eliminate copying of data from the input to the output. This is particularly useful when the output is a pointer to a section within an input token of the input stream as copying the data from the input to the output is avoided. A good example of a primitive that can benefit from using an inplace pointer output is the `v_selsub` primitive that selects a subvector out of a vector based on an integer scalar input stream. A version not using an inplace pointer stream is:

```

Name: v_selsub
Type: static
Input: {
    stream float in[M];
    stream int offset;
    int N;
}
Output: {
    stream float out[N];
}
Apply: {
    int 9,g;

    for (g=0; g< granularity;g++) {
        for (i=0; i<N; i++) {

```

```

        out[i] = in[i+offset[i]];
    }
    in += M; out += N;
}
}

```

Using an inplace pointer stream avoids the copy.

```

Name: v_selsub
Type: static
Input: {
    stream float in[M];
    stream int offset;
    int N;
}
Output: {
    inplace pointer stream float out[N] = in;
}
Apply: {
    forward(out, in, in+offset[0], 1);
}

```

The setting of the output pointer to the input pointer is handled by the `forward` function. The `forward` function has the prototype

```
forward(<output_stream>, <input_stream>, void *input_ptr, int
ntokens);
```

where

- `<output_stream>` is the name (without quotes) of the output pointer stream
- `<input_stream>` is the name (without quotes) of the input stream
- `input_ptr` is the pointer that the output stream is to be set to.
- `ntokens` is the number of tokens to be produced into the output stream.

Note in the example the second parameter `in` is the name of the handle to the input stream and in the third parameter `in` is the pointer to the beginning of the input stream whose handle is `in`. The example could just as well have been:

```
float *a = in+offset[0];
forward(out, in, a, 1);
```

The second parameter must be `in` but the third parameter can be any valid pointer that points offset from the input pointer `in`. The only requirement on the pointer is that the output token must be completely within the input token. So for the above example we must have:

```
in <= a < a+N <= in+M.
```

As with non-inplace pointer streams, the Gedae compiler guarantees that the primitive producing the pointer stream fires at a granularity of 1. As a result for static pointer streams the parameter `ntokens` is redundant as the number of tokens must be the static produce amount of the output. For example if the output is declared as:

```
inplace pointer stream out[N](4) = in;
```

Then because the produce amount is 4 the forward function must be

```
forward(out, in, in_ptr, 4);
```

However pointer streams can be dynamic in which case the number of tokens passed to forward is determined at runtime. For example if the pointer stream output is declared as:

```
dynamic inplace pointer stream out[N](4) = in;
```

Then the number of tokens passed to forward can be any value from 0 to 4 as calculated by the Apply method. Note that the produce function should not be called on dynamic inplace pointer streams as the produce amount is set by the call to forward. It is also legal for inplace pointer streams to be inplace with dynamic or nondet input streams.

For outputs that are inplace with a single input the first parameter is also redundant. However pointer outputs can be declared to be inplace with multiple inputs. For example

```
Input: {
    stream float a[N];
    stream float b[N];
    stream int c;
}
Output: {
    inplace pointer stream out[N] = a,b;
}
Apply: {
    if (*c) {
        forward(out, a, a, 1);
    } else {
        forward(out, b, b, 1);
    }
}
```

Again note that the second parameter indicates the name of the input to be forwarded and the third parameter indicates the pointer to be forwarded. In this case since the pointer is

just the beginning of the input buffer the names passed to forward for the second and third parameters are the same, but their meanings are different.

Note that the list `a, b` in the inplace declaration indicates that `out` can be inplace with either `a` or `b` and the choice will be made dynamically at runtime by the primitive `Apply` method. In the above example the primitive forwards either `a` or `b` to the output depending on the value of the condition variable `c`.

An inplace pointer stream may be inplace with a family of inputs. In this case the primitive will forward a pointer from one of the family members to the output on each execution of the `Apply` method. The family member will be dynamically chosen at runtime. For example:

```
Input: {
    stream float [F]in;
    stream int c;
}
Output: {
    inplace pointer stream out[N] = in;
}
Apply: {
    forward(out, in[*c], in[*c], 1);
}
```

Multiple inplace pointers may be inplace with the same input. This feature can be used to partition a single token into multiple output tokens. For example a matrix can be partitioned into multiple submatrices as:

```
Name: m_rpart
Type: static
Input: {
    stream float in[R][C];
}
Output: {
    inplace pointer stream float [F]out[R/F][C] = in;
}
Apply: {
    int f;
    for (f=0; f<F; f++) {
        forward(out[f], in, in+f*R*C/F, 1);
    }
}
```

Limitation: Currently Gedae does not check that the multiple outputs do not overlap. If they do it is currently up to the user to insure that the primitive following the overlapping data do not modify the data. For example an inplace primitive with an `Apply` method that

modifies the data should not immediately follow a primitive that partitions the data into overlapping outputs.

Inplace pointer outputs are assumed to not be corrupting. That is the values that the pointers point to should not be modified. Gedae uses this assumption to avoid adding copy primitives to fix inplace problems. Thus inplace pointer outputs should only be used to decompose data but not modify it. Modifying the actual values is an error that Gedae will not detect and can cause unpredictable results.

Note if an output pointer stream is to be set to point to input memory it is necessary to use the inplace modifier. If the inplace modifier is not used Gedae does not know that the input memory is still in use after the primitive fires and may assign other primitive outputs to this memory area. This type of problem causes the primitives using the pointer output data to operate on data that has been corrupted.

Inplace Input Pointer Streams

A user may declare an input to be a pointer stream and that pointer stream can be inplace with an output stream. The primitive Apply method can then set the input pointer to point to a memory area within the output stream. For example a primitive that is the opposite of the `m_rpart` is the `m_rnconcat`. This primitive concatenates a family of input matrices into a single output matrix.

```
Name: m_rnconcat
Type: static
Input: {
  pointer stream float [f:F]in[R[f]][C];
}
Output: {
  inplace stream float out[sum(R,F)][C] = in;
}

Apply: {
  int f;
  int Rsum = 0;
  for (f=0; f<F; f++) {
    set_ptr(in[f],out+Rsum*C);
    Rsum += R[f];
  }
}
```

The primitive uses the `set_ptr` function with just two variables to set the input stream to the output pointer. In this context the `set_ptr` function has the prototype

```
set_ptr(<input_stream>,void *output_ptr)
```

Where

`<input_stream>` is the name of the input stream whose pointer value is being set.
`output_ptr` is the pointer within the output memory to which the input is being set.

Unlike every other primitive type in Gedae a primitive with an input pointer must fire before the primitive driving its input. This ordering is necessary so the input pointer primitive can set the pointer of the output that is driving it. Currently – to avoid circular ordering problems a primitive with an input pointer must have all its stream inputs be pointers.

As with inplace pointer outputs, pointer inputs inplace with an output are assumed to be non-corrupting. It is an error – that Gedae will not detect – for a primitive with an inplace input to modify the values of the data being pointed to. Adding such a primitive will cause unpredictable results.

Inplace Tiled Pointer Streams

A tiled stream may also be declared to be an inplace pointer. In fact this is one of the main uses of the tiled stream notation. An example is the `mt_cpartN` primitive

```
Name: mt_cpartN
Type: stream
Input: {
    stream float in[Rs:R][Cs:C](iterate = N);
    int N;
}

Local: {
    int which;
}

Output: {
    inplace pointer stream float out[Rs:R][Cs/N:C] = in;
}

Start: {
    which = 0;
}

Apply: {
    forward(out, in, in+which*(Cs/N), 1);
    which++;
    if (which == N) {
        which = 0;
    }
}
```

```
}
```

This primitive partitions the input matrix column wise into N equal sized pieces and time multiplexes them onto the output stream. Note that because the partitioning is column wise we must use a tiled stream if the operation is to be inplace. Another example is the `mt_rpart` which is the tiled version of the `m_rpart` described earlier. The `mt_rpart` is:

```
Name: mt_rpart
Type: static
Input: {
    stream float in[Rs:R][Cs:C];
}
Output: {
    inplace pointer stream float [F]out[Rs/F:R][Cs:C] = in;
}

Apply: {
    int f;
    for (f=0; f<F; f++) {
        forward(out[f], in, in+f*(Rs/F)*C, 1);
    }
}
```

Setting of Tiled Stream Dimensions

The following rules dictate how dimensions are set for tiled streams.

- When a tiled stream is connected to a tiled stream then the tiled and normal dimensions of both must be equal. This is the obvious rule for propagating dimensions.
- When a non-tiled stream is connected to a tiled stream then the non-tiled dimensions are equal to the tiled streams tiled dimensions. This rule makes sense as the only values that are valid in a tiled stream are the tiled values. Gedae automatically inserts a tiled to non-tiled conversion primitive if the tiled and normal dimensions are not equal.
- When a non-tiled stream is connected to a non-pointer tiled stream then the tiled stream's normal dimensions must be equal to the non-tiled stream dimensions. In this case the non-tiled stream dictates the matrix size in which the tiled stream is embedded.
- When a non-tiled output stream is not connected then the normal dimension is set to the tiled dimension. This is really the same thing that would occur if the tiled streams output was connected to a non-tiled stream.
- If a pointer tiled stream is connected to a tiled stream then the pointer streams normal dimension is only equal to the tiled streams dimension if there is not a contradiction with one of the other rules.

- An input tiled stream's tiled dimensions are set by the source the tiled stream is connected to. (This is the same as the way normal dimensions are set for non-tiled streams).
- An output tiled streams tiled dimensions are set by equations that are in terms of input dimensions, family indices and input parameters. (This is the same as the way normal dimensions are set for non-tiled streams).
- Both input and output tiled stream normal dimensions are set by virtue of what they are connected to. Neither may be set by equations based on parameters.
- Inplace tiled streams must have the same normal dimension names.

The rules leaves open the possibility that a tiled stream pointer normal dimension is not equal to the non-tiled streams dimension. This is an important case that Gedae handles by automatically inserting a tiled-to-nontiled conversion primitive. A typical graph using tiled streams connects a source non-tiled stream to a series of primitives that have tiled inputs and pointer tiled outputs. The series of boxes decomposes the input matrix into smaller and smaller tiles. The last pointer box in the chain is connected to a non-tiled series of primitives that does the actual processing of the tile. The last non-tiled primitive output is connected to a tiled pointer input that inserts the tiles into the large matrix. A series of tiled boxes with inplace pointer inputs builds larger and larger subtiles. The large matrix size is dictated by the calculated tile size of the last tiled box in the chain.

10 Unmapped Memory

Memory that a processor can directly dereference is called **mapped** memory. For example if `float *out` is a mapped memory pointer then the expression `*out` is valid and will return the value of `out`. Memory that can only be accessed by copying it using a functional interface is called **unmapped** memory.

Some processors provide large areas of unmapped memory. For example the Cell/B.E. allows each SPU to access only 256kbytes of mapped memory but allows to many gigabytes of unmapped memory. In Gedae some BSPs declare different unmapped memory types and Gedae provides generic BSP functions for moving data between unmapped and mapped memory. A primitive can declare that one or more of its inputs or outputs are unmapped memory using the `unmapped` keyword as for example:

```
Name: mu_noop
Type: stream
Input: {
    stream unmapped float in[R][C];
}
Output: {
    inplace stream unmapped float out[R][C] = in;
}
```

The above box declares its inputs and outputs to be unmapped. This declaration is propagated through all non-corrupting inplace primitives that the `mu_noop` input/outputs are connected to. Gedae stops propagating the unmapped property when it encounters a primitive I/O that is not inplace or that is corrupting. In that case Gedae automatically inserts a primitive to move data from unmapped to mapped memory. If the unmapped memory is tiled Gedae automatically inserts a primitive that moves the data from a subtile of the unmapped memory into mapped memory.

Below is an example taken from the `internal/getu` primitive that gedae automatically inserts to move data between non-tiled unmapped to mapped memory. The `e_getu` function – which is part of the BSP for each processor that supplies unmapped memory – is used to move data between unmapped and mapped memory.

```
Name: getu
Type: stream
Input: {
    stream unmapped void in;
}
Output: {
    stream void out;
}
Include: {
```

```
#include <e_getu.h>
}
Apply: {
    e_getu(out, in, size(out));
}
```

Users can write their own primitives that have unmapped inputs/outputs and mapped outputs/inputs and that move data between these inputs and outputs. A complete set of unmapped to mapped data transfer functions is described in the **Gedae Primitive Function Reference Manual**.

11 Persistent Memory

Declaring an output stream persistent reserves the memory so that the data written to it in one execution of a schedule remains valid for successive executions. Only the primitive that declares the memory area persistent and any destination primitives (including inplace destinations that modify the memory area) can have access to it. An output stream can be declared persistent using the `persistent` keyword as:

```
Output: {  
    persistent stream float X[N];  
}
```

This declaration reserves a memory area of size `N*sizeof(float)` bytes that will only be available to the destination primitives. A persistent output acts much like a primitive `Local` except that it can be shared and modified by multiple primitives. A persistent output should only be part of a primitive that executes with a granularity of 1.

12 Segmented Data Flow

TBD

Segmented Outputs

Segmented Parameters

Exclusive Outputs

External State

13 Runlength Encoded Streams

Runlength encoded streams allow you to specify how long, in number of tokens, a stream keeps its value. They are useful if the value of a variable rarely changes, for example, if it changes only during the transition between modes. If a stream has values {31, 31, 31, 42, 42, 27, 27, 27, 27, 27, 27, ...}, then the runlength encoded stream is {(31,3), (42,2), {27,7), ...}.

Let's look at an example where a standard scalar stream is added to a runlength encoded stream. The inputs and outputs are declared as:

```
Input: {
    stream float in;
    stream encoded float k;
}
Output: {
    inplace stream float out=in;
}
```

where `k` is the runlength encoded parameter. To decode this special parameter, use the `void *decode(void *q, int *avail, int amount)` function. This function takes input parameters `q` and `amount`. The parameter `q` is the encoded input queue, and the parameter `amount` is the maximum number of tokens to be read. The function returns a pointer to the value of the token and returns the token's runlength in the parameter `avail`.

To see how this function is used, let's look at the `Apply` method for adding these two inputs together.

```
Apply: {
    int g;
    for (g=0; g<granularity; ) {
        int n;
        float nextk = *(float *)decode(k, &n, granularity-g);
        e_vsadd(in+g, 1, nextk, in+g, 1, n);
        g += n;
    }
}
```

The `decode` function is called in each iteration of the for-loop. The variable `g` keeps account of how many tokens have been used; thus, `granularity-g` tokens are remaining to be computed during this call of the `Apply` method. The call to `decode` returns the next value on the `k` stream along with its runlength (at most, `granularity-g`). With the value of `k` and its runlength, we can call `e_vsadd` to perform the addition.

The function `void encode(void *output, void *next_value, int n)` encodes a runlength encoded stream. To see how the `encode` function is used, let's look at the box that turns a standard scalar stream into a runlength encoded stream, that is, `embeddable/stream/encode`. In the example below, the input is a scalar stream, and the output is a runlength encoded stream:

```
Input: {
    stream float in;
}
Output: {
    stream encoded float out;
}
```

To perform the transformation, test to see if the new token is the same as the previous one on the queue. If they are different, then the previous token is encoded on the runlength encoded output. If they are the same, then a counter is incremented, and the next token is considered. At the end of the `Apply` method, the current value is encoded with the current value of the counter (the counter is not retained between calls to the `Apply` method). Below is the full `Apply` method:

```
Apply: {
    int g;
    float nextout = in[0];
    int count = 1;
    for (g=1; g<granularity; g++) {
        if (in[g] == in[g-1]) count++;
        else {
            encode(out, &nextout, count);
            nextout = in[g];
            count = 1;
        }
    }
    encode(out, &nextout, count);
}
```

14 Cyclic Boxes

All the boxes described up to this point are of `Type: static`. This type of box – with dynamic and nondeterministic queues, or static queues with fixed produce, consume, overlap and hold, etc. – provides you with a great deal of flexibility and power. However, there are times when the static implementation of an operation is possible but inefficient.

One example of when the static box implementation is naturally inefficient is when produce and consume amounts must be set high in order to accommodate a redistribution of data. For example, a commonly used box in the Gedae library is the `mux` box (abbreviation for Multiplexer). A `mux` box receives a family of tokens and turns them into a single output stream. This operation is illustrated in Figure 19: a family of three input streams is received, and the tokens are placed sequentially on the output stream.

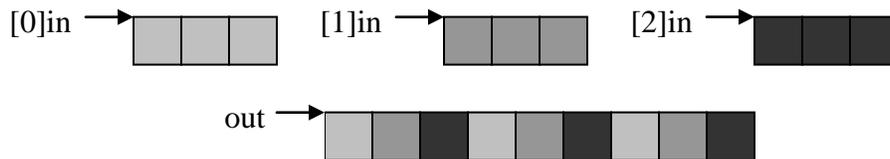


Figure 19 – The operation of a mux box

Implementing the `mux` box for a floating-point stream is simple:

```
Name: mux
Type: static
Input: {
    stream float [M]in;
}
Output: {
    stream float out(M);
}
Include: {
#include <e_vmov.h>
}
Apply: {
    int j;
    for (j=0; j<M; j++)
        e_vmov(in[j],1,out+j,M,granularity);
}
```

The input stream is a family of size `M`; therefore, the output stream should have a produce amount of `M`. In the `Apply` method, `granularity` tokens are moved from family member `j` to the correct place on the output using a stride of `M`.

The above implementation of a `mux` box is short and elegant. However, it places a requirement on the input stream that each family member must have `granularity` tokens present in order for any tokens to be produced on the output. The nature of this inefficiency is in the latency of the box: if family member 0 quickly provides a token for consumption, but family member 1 must finish a more lengthy computation before providing a token, then the `mux` box will wait for family member 1 before member 0's token can be copied to the output stream.

If a latency of this sort is an issue, then **cyclic boxes** can be used to create a more efficient implementation. Cyclic boxes are not static boxes; they have their own type, `Type: cyclic`. They also have two new methods that static boxes do not have. Let's re-implement the above `mux` box as a cyclic box.

A cyclic box is so named because it cycles through several states. For a `mux` box, the states can be described by what input token was last copied to the output. When the box copies a token from family member 0 to the output, the next action is to copy family member 1's token to the output. The state after copying family member 0 is different than the state after copying family member 42. After copying family member 42, the next action will be to copy family member 43's token.

Thus, for a `mux` box, the number of states is the same as the number of family members. To specify the number of states, we must include a `Length` method in our cyclic box. The `Length` method returns the number of states. If there are M family members, then the `Length` method is:

```
Length: {  
    return M;  
}
```

Now that the length of the cycle has been defined, we must create the input and output declarations so that `Gedae` knows how many tokens are produced in each state.

In our static version of the `mux` box, each time the box executes, it produces $G * M$ tokens on the output (where M is the family size and G is the granularity). If there are four family members on the input stream, then G tokens are consumed on each input family member, or, in other words, $\{G, G, G, G\}$ tokens are consumed. With a cyclic box, this consumption and production of tokens is broken into stages. For a `mux` box with a family of size four, the stages are:

0. $\{0, 0, 0, 0\}$ tokens have been consumed, 0 token has been produced,
1. $\{1, 0, 0, 0\}$ tokens have been consumed, 1 token has been produced,
2. $\{1, 1, 0, 0\}$ tokens have been consumed, 2 tokens have been produced,
3. $\{1, 1, 1, 0\}$ tokens have been consumed, 3 tokens have been produced,
4. $\{1, 1, 1, 1\}$ tokens have been consumed, 4 tokens have been produced.

The M -element vector defines at each stage how many tokens have been consumed in the cycle. This vector is used to specify the consume amount for the family input of the mux box – the consume amount of the i -th family member is element i of this vector.

Study the previous four-member example to see how this vector defines the consume amount of the input. At stage 2, the first two inputs have had tokens consumed, but the second two inputs are not required to have any tokens available for consumption. The latter two inputs, during this stage, have a consume amount of 0; they are not required to have any tokens. Likewise the first two inputs, during this stage, have a consume amount of 1; they must provide a token for consumption by this stage in the cycle.

To form this vector and declare the input with its consume amount set to the vector, declare a local vector to store the consume amounts, and then set its values in the `Cycle` method. If the declaration `int N[M];` is added to the `Local` section of the box (where M is the family size of the input), the following `Cycle` method will construct the N vector:

```
Cycle: {
    int i;
    for (i=0; i<M; i++) {
        N[i] = (firing > i) ? 1 : 0;
    }
}
```

The `Cycle` method uses a built-in variable `firing` to specify which stage of the cycle its calculation relates to. When `firing=0`, $N[]=\{0, 0, 0, 0\}$, when `firing=1`, $N[]=\{1, 0, 0, 0\}$, etc. With this `Cycle` method specified, the family input is declared as:

```
Input: {
    stream float [i:M] in(N[i]);
}
```

Family member i has consume amount $N[i]$, and $N[i]$ has a different value for each stage in the cycle.

Now that the input has been declared, the output must also be declared in a special manner for cyclic boxes. In the static version of the mux box, the produce amount of the output is the same as the family size. In the cyclic version of the box, the produce amount is the sum of the N vector for that stage. The N vector, as described above, specifies how many tokens are consumed on each input up to that stage in the cycle. Each token consumed is produced on the sole output stream; thus, the number of output tokens produced up to that stage in the cycle is the sum of the N vector for that stage.

To specify the produce amount for the output, add a declaration for `int Sum;` to the `Local` section of the box, and then extend the `Cycle` method to tally the `N` vector in the `Sum` variable, as follows:

```
Cycle: {
    int i;
    Sum = 0;
    for (i=0; i<M; i++) {
        N[i] = (firing > i) ? 1 : 0;
        Sum += N[i];
    }
}
```

In actuality, due to the behavior of the `mux` box, the `Sum` is the value of the variable `firing`. With the produce amount of the output specified in the `Cycle` method, the output stream is declared as:

```
Output: {
    stream float out(Sum);
}
```

The value of `Sum` is different for each stage in the cycle; in fact, `Sum` increments between stages as each stage adds another token to the output.

Now that the input and output streams have been declared, the `Apply` method can be implemented. Like static boxes, cyclic boxes have `Apply` methods and must be written to process all the tokens on the queues. The number of tokens is defined by the `granularity` variable.

The `Apply` method must be written to duplicate the state behavior of the `Cycle` method. In the `Apply` method for the cyclic `mux` box, we will use the local variable `where` to specify which family member to read from first. The declaration `int where;` is added to the `Local` section of the box, and a `Reset` method is added to initialize the variable like it would be for a static box.

The `Reset` method is:

```
Reset: {
    where = 0;
}
```

|

The Apply method is:

```
Apply: {
    int j;
    for (j=0; j<M; j++) {
        int k = where + j;
        int len = (granularity + M - j - 1)/M;
        if (k >= M) k -= M;
        e_vmov(in[k],1,out+j,M,len);
    }
    where = (where + granularity) % M;
}
```

When the Apply method is called, we loop through each family member of the input, starting at the member pointed to by `where`. For each family member, the number of tokens to copy is calculated, and then the copy is performed using a stride of `M`, the family size. Finally, the value of `where` is updated in preparation for the next execution of the box. The full implementation is shown in Figure 20.

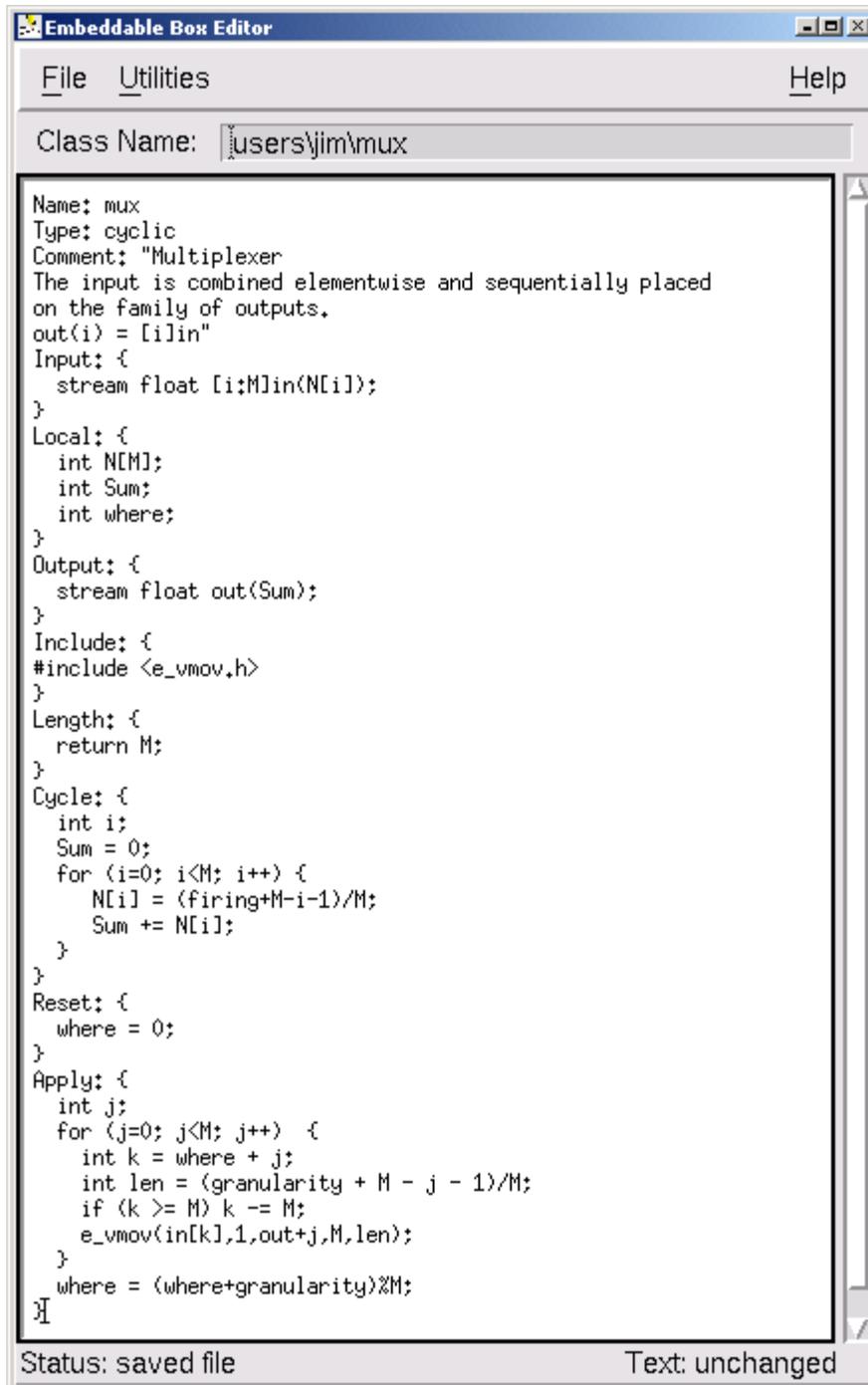
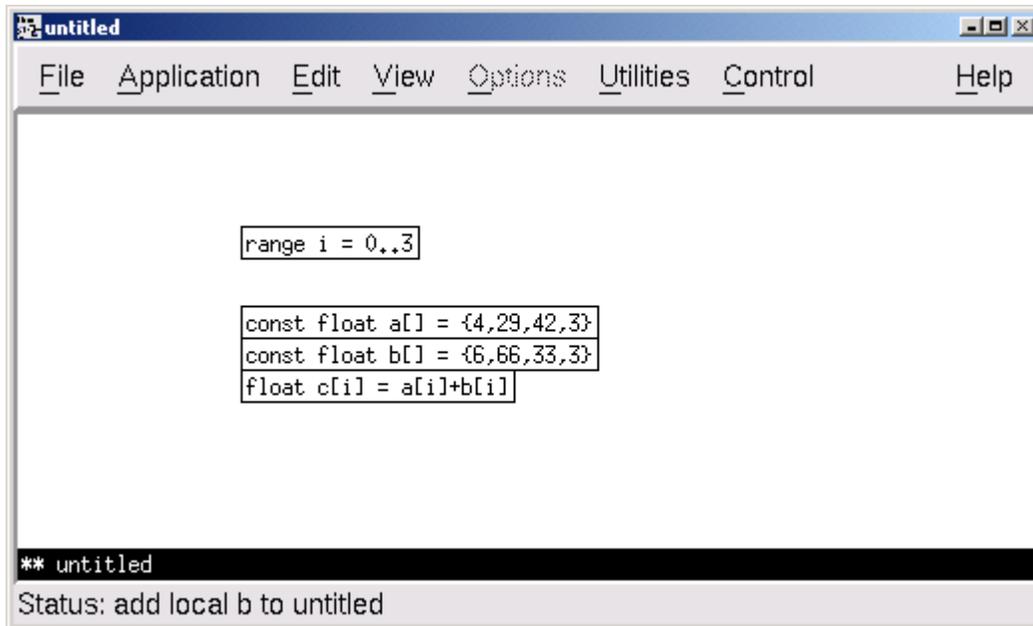


Figure 20 – The implementation of the mux box

15 Eval and Trigger Boxes

Streams are at the heart of a Gedae graph. All box types discussed in previous chapters must have a stream. Static, nondet, and cyclic boxes cannot be written solely to use parameter inputs and outputs. If a box is needed that has only parameter inputs and outputs, then an **eval** or **trigger** box must be used. However, these boxes run only on the host; they cannot be mapped to other processors as they are only intended to simplify the process of calculating parameters.

Some parameter calculations are implemented entirely on the canvas. For example, in Figure 21, vectors `a []` and `b []` are available on the canvas. If we wish to add these two vectors together, then we add a declaration of a new vector `c []`, and use the “+” operator to add the two vectors together.



```
range i = 0..3

const float a[] = {4,29,42,3}
const float b[] = {6,66,33,3}
float c[i] = a[i]+b[i]
```

** untitled
Status: add local b to untitled

Figure 21 – Some operations are implemented on the canvas

Most elementwise arithmetic and basic math is done on the canvas. However, if a more complicated algorithm needs to be done on parameter data, then it cannot be done on the canvas via the declaration. For example, if the Fast Fourier Transform needs to be done on a parameter vector, then there is no built-in FFT function that can be called in the data declaration. Another example is the sum of the squares of the vector elements. In Figure 21, if the sum of squares of `b []` needs to be calculated for a parameter input to a box, then we could write out the full equation by declaring a data element `b_ssq = b[0]*b[0] + b[1]*b[1] + b[2]*b[2] + b[3]*b[3]`; however, it would be much more convenient if the sum could be calculated for any vector of any length.

Eval Boxes

To perform a complex parameter calculation such as summing the squares of elements of an input vector, a box of Type: `eval` should be used. **Eval** boxes have parameter inputs and parameter outputs; the Input and Output sections of the box will be similar to those of static boxes, but only parameters will be included. The boxes do not have granularities, Apply methods, or Reset methods because they only act on one set of input values. Instead, an eval box has an Eval method.

For a box that calculates the sum of a vector's elements, the input is a vector and the output is a scalar. These variables are declared as:

```
Input: {
    float In[N];
}
Output: {
    float Out;
}
```

If the parameters have default values, then an Init method (similar to those for static boxes) can be used. The Init method is not used for this sum box. All that is needed is the Eval method that calculates the sum as shown in Figure 22.

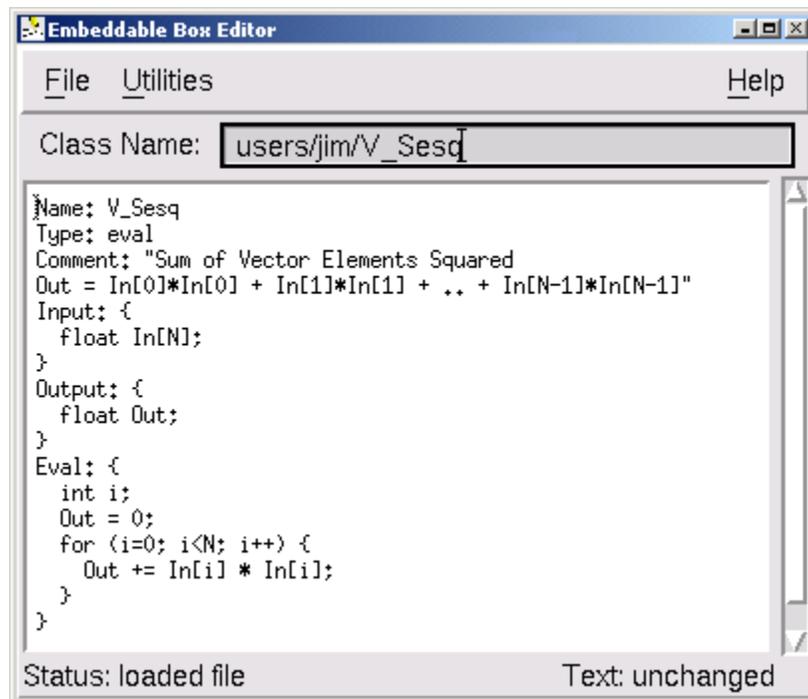


Figure 22 – Implementation of parameter sum of vector elements squared

Trigger Boxes

Trigger boxes are an extension of eval boxes that allow you to control the sequence of execution. When eval boxes are used the outputs are always up-to-date with respect to the inputs; if an input value changes, then the output value is recalculated. Trigger boxes give you more control over when and how the output value is recalculated.

The eval box for summing a vector's elements squared in the previous section will recalculate the output value whenever the value of the input changes. If a change to the input does not necessarily result in a change to the output, then the box should have `Type: trigger`. Trigger boxes do not have `Apply` methods or `Eval` methods; they have `Trigger` methods. (Trigger Boxes can also have `Reset` methods, `Init` methods, `Local` methods, `Destroy` methods, `Save` methods, and `Restore` methods.)

To change the `V_Sesq` box in the previous section to a trigger box, three changes need to be made. First, the `Type` field must be changed to `trigger`. Second, the input parameter must be declared as a `trigger`:

```
Input: {
  trigger float in[N];
}
```

Trigger boxes can have both trigger and non-trigger inputs, but only changes to trigger inputs will cause the `Trigger` method to be called. Outputs are not declared as triggers. The third change is that the output `Out` must be **pushed**. When an output is pushed, other trigger and eval boxes will receive the output's updated value. The `push` function doesn't return until the effect of the push (including calling of other trigger boxes which in turn may push variables) has finished propagating through the graph. The output `Out` is pushed by simply calling `push(Out)`.

With these three changes, the altered `V_Sesq` box can be used but performs the same operation as the original eval box. To utilize the functionality of triggers, put a condition on whether the output is pushed. For example, to push the output value only if the sum of the vector's elements squared is greater than one, replace the call to `push(Out)` in the `Trigger` method with the following:

```
if (Out > 1) push(Out);
```

If the new value of the sum is not greater than one, then the old value of `Out` is retained. The capability of conditionally pushing parameters is useful for implementing state machine-like functionality in a graph.

The `Trigger` method is called when a new value is available on a trigger input. If multiple inputs to a box are triggers, then it could be useful to know which trigger input

has received a new value. The function `int dirty(trigger input)` can be called inside the `Trigger` method to determine if a trigger input has received a new value.

The `push` function takes any number of inputs to allow multiple outputs to be pushed simultaneously. For example, if the outputs `Out0`, `Out1`, and `Out2` are to be pushed simultaneously, then `push(Out0, Out1, Out2)` will push all three outputs. There is no requirement that all output parameters be pushed simultaneously. The `push` function can be called numerous times in the same box, allowing different outputs to be pushed under different conditions.

Static boxes can also have parameter outputs. The `push` function is used in the same manner as it is used in trigger boxes to inform other boxes of updated values.

GUI Trigger Boxes

Trigger boxes are useful in constructing GUIs. The GUI library supplied with Geda is constructed mostly from trigger boxes. To accommodate their use in constructing GUIs, trigger boxes have several methods that other types of boxes do not. While these methods are useful in GUI trigger boxes, they can be used in any trigger box.

Because GUIs create shells that reside in memory, Destroy methods are available for use in trigger boxes. For example, if a shell is created in the GUI box, a `Local` section variable `shell` can be declared to store the variable, and a suitable Destroy method may be:

```
Destroy: {
  if (shell) {
    XtDestroyWidget(shell);
    shell = 0;
  }
}
```

Similarly, any memory or other resources that are allocated can be freed here.

`Save` and `Restore` methods are also specific to trigger boxes. In the display boxes included in the Geda library many internal parameters, such as the location of the display on the screen and the size of the display, are saved in the graph's parameter file. The saving of these parameters allows the displays to look the same each time the graph is loaded and run. To be able to save these parameters (without including them as inputs to the box), `Save` and `Restore` methods are used.

Utility functions are available for saving and restoring each required data type (`int`, `float`, `double`, and `string`) as shown in Table 2. If the width, height, x location, and y location of

a display are to be stored in the parameter file, declare variables for each item in the Local section of the box, then set the Save and Restore method as:

```
Save: {
    save_int(x);
    save_int(y);
    save_int(width);
    save_int(height);
}
Restore: {
    restore_int(x);
    restore_int(y);
    restore_int(width);
    restore_int(height);
}
```

Type	Save	Restore
int	save_int(int x);	restore_int(int *x);
float	save_float(float x);	restore_float(float *x);
double	save_double(double x);	restore_double(double *x);
string	save_string(char *s);	restore_string(char **s);

Table 2 – Functions for saving and restoring values in the parameter file

The order that the variables are saved and restored in should be the same, that is, if x is saved before y, then x should be restored before y. These methods can contain any C-code, so it is possible to retrieve these values from the GUI structures before saving them in the Save method, and it is possible to manipulate these values and copy them to local variables after retrieving them in the Restore method.

The final type of method specific to trigger boxes is the Callback method. GUI buttons and other objects have callback functions that define what happens when certain actions are performed on the object. For example, a button will have a callback function that is performed when the button is pressed. To define a callback function in a trigger box, you can define a Callback method. For example, assume a GUI has two buttons and each time button 1 is pressed, the GUI prints to the terminal “Button 1 was pressed” and each time button 2 is pressed, the GUI prints “Button 2 was pressed”. The Callback method is:

```
Callback: {
    int i = (int)calldata;
    printf("Button %d was pressed\n", i);
}
```

This method creates a callback function `void Callback(Widget self, void *calldata)` that can be called from any other method in the box. Usually this function will be used with the GUI function `XtAddCallback` to associate the Callback

method with a certain button or action. Unlike other methods, the Callback method can have any name; any method or field that has a name not known to Gedae is assumed to be a Callback method.

16 Typedef Boxes

In Chapter 2, the possible data types of Gedae streams are discussed, including the capability to create streams of custom structures. Gedae also includes the capability to create structures of streams. Such structures are made using typedef boxes.

Aside from having `Type: typedef` and a `Name` field, a typedef box only has one other field called the `Input` field. Each stream that is an input to the typedef box is bundled together into an output structure of streams. For example, if we want to create a structure containing three streams for storing 3-D coordinates, then we could construct the following typedef box using the base type of `void`:

```
Name: coord3d
Type: typedef
Input: {
    void x;
    void y;
    void z;
}
```

Let's assume the above box is stored in `FGlibraries/boxes/types/coord3d`. This type can now be used with the `void` type to create new types. For example, if we want to create a structure for storing pyramids with four points, then we can define:

```
Name: pyramid
Type: typedef
Input: {
    types/coord3d p0;
    types/coord3d p1;
    types/coord3d p2;
    types/coord3d p3;
}
```

We simply refer to the `coord3d` type by its location under `FGlibraries/boxes`.

Typedef boxes added to the canvas appear as constructors. Constructor boxes take the component streams as inputs and create a structure of streams as the output. Destructor boxes do not have to be created separately. Instead, add a constructor box to the canvas, and while holding the `Ctrl` key down, double click on the name of the box. The constructor and destructor of the `pyramid` type are shown in Figure 23.

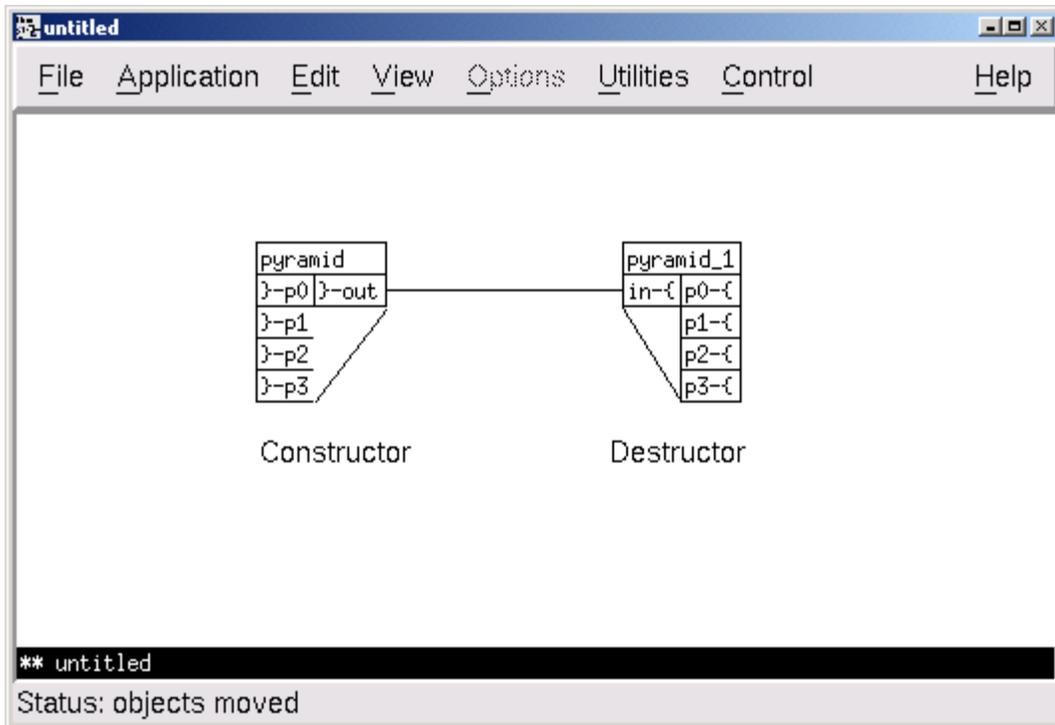


Figure 23 – The typedef box defines both the constructor and the destructor

Appendix – Suggested Style Guide

Gedae has several style conventions that are designed to help make primitives easier to read and understand.

Naming Primitives

The name of a primitive should indicate its primary token and data type through a prefix. If the box performs a conversion in type, then the box should have a prefix that indicates the primary type of the input and a suffix that indicates the primary type of the output. Prefixes and suffixes are separated from the rest of the name by an underscore (“_”). The abbreviations for token and data types are shown in Table 3. Note that a suffix or prefix is not used for scalar floating point streams unless the box performs a conversion of type.

Table 3 – Abbreviations for Token and Data Types

	Scalar	Vector	Matrix	Var-Vector	Var-Matrix
Float	none (s)	v	m	vv	vm
Complex	x	vx	mx	vvx	vmx
Integer	i	vi	mi	vvi	vmi

Naming Variables

Names of variables for streams and triggers are not capitalized. This practice includes input and output streams, length streams for variable vectors and variable matrices, trigger inputs, and range variables over families.

Names of variables for parameters are capitalized. This practice includes input and output parameters, local variables, array lengths, maximum sizes for variable vectors and variable matrices, and family sizes.

Comments

The first line in a primitive’s comment should be a short description of the box’s purpose. The next line should be a formula or pseudo-code that precisely defines the box’s algorithm. If further discussion is necessary, then include it after the formula.

Index

- amount, 36, 37
- Apply method, 8
- avail, 36, 37
- box, 6
- boxes
 - cyclic, 62
 - eval, 68
 - mux, 61
 - primitive, 6
 - trigger, 69
 - typedef, 73
- built-in functions
 - amount, 36
 - avail, 36, 37
 - consume, 36, 37
 - decode, 59
 - dirty, 70
 - encode, 60
 - produce, 37
 - push, 69
 - save_double, 71
 - save_float, 71
 - save_int, 71
 - save_string, 71
 - size, 14
- built-in variables
 - firing, 63
 - granularity, 8
 - queues_ready, 37
- Callback method, 71
- Comment field, 7
- complete merge, 38
- complex, 12
- consume, 36, 37
- consumed, 20
- convert
 - scalars to vectors, 20
 - variable vectors to scalars, 35
- Creating a Primitive, **6–12**
- Cycle method, 63
- cyclic boxes, 62
- Cyclic Boxes, **61–66**
- data flow parameters
 - delay, 27
 - hold, 26
 - overlap, 24
- data types, 12
 - complex, 12
 - float, 12
 - int, 12
- Data Types, **12**
- decode, 59
- delay, 27
- Delay, **27–30**
- Destroy method, 70
- dirty, 70
- dynamic, 20
- Dynamic and Nondeterministic, **35–42**
- dynamic queue, 35
- Dynamic Queues, **35–36**
- E_function, 10
- E_functions, **10–11**
- encode, 60
- Eval and Trigger Boxes, **66–72**
- eval boxes, 68
- Eval Boxes, **68**
- Eval method, 68
- execute, 35
- Families, **31–34**
 - output vectors of different lengths, 33
- fields
 - Comment, 7
 - Name, 7
 - Type, 7
- fire, 35
- firing, 63
- Fixed Inplace, 46
- float, 12
- granularity, 8, 35
- granularity loop, 8
- GUI, 70
- GUI Trigger Boxes, **70–72**
- hold, 26
- Init method, 11, 68
- inplace, 9, 46
- Inplace Input Pointer Streams, 51
- Inplace Output Pointer Streams, 47
- Inplace Streams, **9–10**

- Input, 7
- Input/Output modifiers
 - dynamic, 35
 - encoded, 59
 - inplace, 9, 46
 - nondet, 37
 - parameter, 11
 - stream, 7
 - trigger, 69
- int, 12
- Introduction, 5
- Iterate, 29
- latency, 62
- Length method, 62
- Local, 22
- Local Variables and Reset Methods, **22–24**
- mapped, 55
- Matrices, **15**
- matrix, 15
- methods, 6
 - Apply, 8
 - Callback, 71
 - Cycle, 63
 - Destroy, 70
 - Eval, 68
 - Init, 11, 68
 - Length, 62
 - Reset, 23
 - Restore, 70
 - Save, 70
 - Trigger, 69
- multiple-dimensional, 32
- mux box, 61
- Name field, 7
- named parameter notation, 41
- nondet, 37
- nondeterministic queues, 36
- Nondeterministic Queues, **36–42**
- nondeterministic queues are grouped
 - into a family, 40
- no-op, 21
- Optional Inplace, 46
- Output, 8
- overlap, 24
- Overlap and Hold, **24–27**
- parameter, 11
- Parameter Inputs, **11**
- persistent, 57
- Pointer Streams, 43
- position dependent notation, 41
- primitive, 6
- Primitive
 - creating a, **6–12**
 - simple, 7–9
- produce, 36, 37
- produced, 20
- push, 69
- pushed, 69
- queues_ready, 37
- Reset method, 23
- Restore method, 70
- Runlength Encoded Streams, **43–60**
- Save method, 70
- save_double, 71
- save_float, 71
- save_int, 71
- save_string, 71
- schedule memory, 43
- sections
 - Input, 7
 - Local, 22
 - Output, 8
- Simple Primitive, **7–9**
- size, 14
- sliding window average, 24
- static, 20
- stream, 7
- subgraph, 6
- threshold, 35
- Tiled Matrix, 17, 52
- Token Types, **13–18**
- Tokens in the Stream, **20–30**
- trigger, 69
- trigger boxes, 69
- Trigger Boxes, **69–70**
- Trigger method, 69
- Type field, 7
- typedef boxes, 73
- Typedef Boxes, **73–74**
- unmapped, 55
- variable matrix, 17

variable vectors, 16
Variable Vectors and Matrices, **16–17**

vector, 13
Vectors, **13–15**