

01000111 01100101 01100100 01100001 01100101
01000111 01100101 01100100 01100001 01100101
01000111 01100101 01100100 01100001 01100101
01000111 01100101 01100100 01100001 01100101

Gedae 4.5.1 Release Notes

June 2004

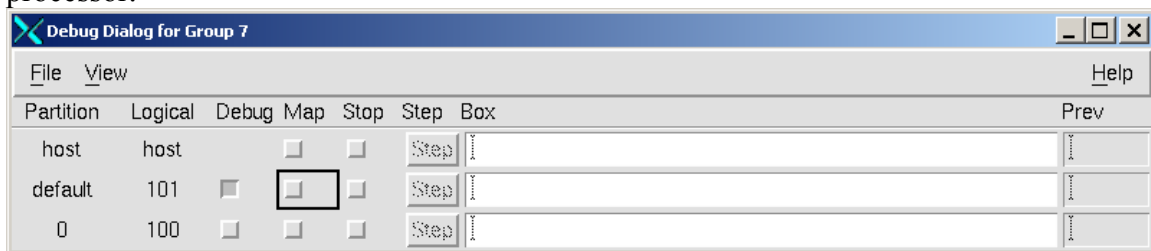
Address: Gedae, Inc.
18000 Horizon Way, Suite 200
Mt Laurel, NJ 08054
Telephone: (856) 231-4458
FAX: (856) 231-1403
Internet: www.gedae.com

1 New Features

Several new features have been added.

Case 1170: Symbolic Debugging Support

New functions have been added to Gedae to enhance symbolic debugging. Symbolic debugging of Gedae applications has always been possible. Some Gedae BSPs provide the ability for a user to start a symbolic debugger by selecting the Debug toggle for the processor.



Alternatively, you can start the symbolic debugger outside of Gedae. Typically a debugger can be started after the target process has begun by passing the target process id to the debugger.

Once in the symbolic debugger, new functions allow you to find the classname, index and the reason the last box fired failed (or not). The information required to fire a function box is called a closure. Each instance of a box in a graph has its own closure. Gedae now provides functions that allow the user to determine the name of the last closure fired and other information about the closure. These functions are:

```
int lastClosureIndex(void) :  
    returns the index of the closure that just fired or is firing.  
char *lastClosureClassname(void) :  
    returns the classname of the closure that just fired or is firing  
char *lastClosureReasonFailed(void) :  
    returns the character string stating why the closure that just fired or is firing has  
    failed, paused or been suspended. If the closure is not in a failed state, then the  
    function returns 0.  
char *indexedClosureReasonFailed(int index) :  
    returns the character string stating why the closure having the given index has  
    failed, paused or been suspended. If the closure is not in a failed state, then the  
    function returns 0.  
char *indexedClosureClassname(int index) :  
    returns the classname of the closure having the given index.
```

An example of how these functions can be used in msdev (the Microsoft Development Studio, the debug environment on Windows platforms) is shown below:

Name	Value
lastClosureReasonFailed()	0x0041e530 "cannot send data"
lastClosureClassname()	0x0041e4f0 "internal/enqueue"
lastClosureIndex()	0x00000009
indexedClosureReasonFailed(14)	0x0041ed44 "resume called during embPause"
indexedClosureClassname(14)	0x0041e4f0 "internal/enqueue"

Watch1 Watch2 Watch3 Watch4

Another enhancement to simplify debugging is that now all code-generated method names are unique. Previously the .c file generated for a primitive with an Apply method would have a method named Apply. All Apply methods were named the same way making it difficult to uniquely name which Apply method the user wanted to set a breakpoint on. Now all the method names are unique and have the form:

<method_name><directory_serial_no>_<primitive_name>

where

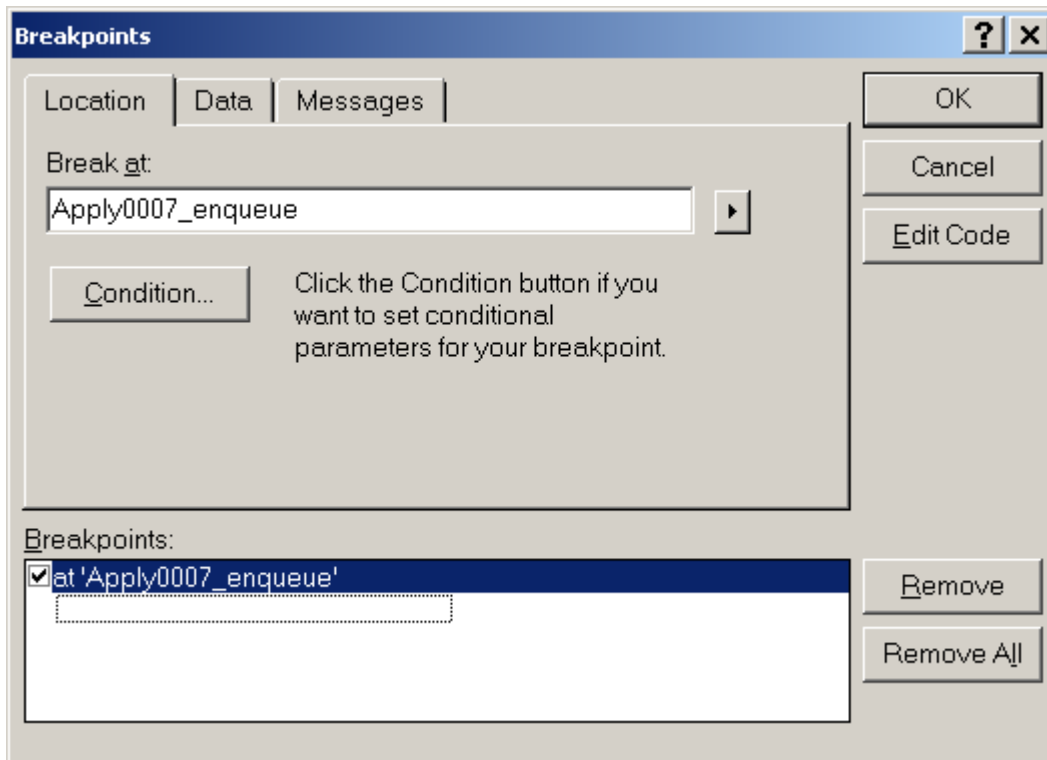
- <method_name> is the name of the method (for example Apply, Reset or Eval)
- <directory_serial_no> is the four digit serial number associated with the directory of the primitive found in FGLibraries. For example, boxes found in library embeddeable/stream will all have the same serial number and this number will be different from the boxes found in embeddable/stream/source. These serial numbers may change after a makeGEDAE CLEAN.
- <primitive_name> the name of the primitive – for example "add".

An example of such a primitive name is Apply0007_enqueue, which is the Apply method for the internal/enqueue primitive. Here the serial number 0007 has been associated with the directory internal.

You can go to the .c file generated for a primitive and find the unique Apply method name for that primitive – in this case Apply0007_enqueue.

```
void I0007_enqueue(void) {
    OCreateStaticBoxClass("internal/enqueue", sizeof(StateRec),
        Offsets, 0,
        0,
        0, 0,
        (void(*) (void*)) Start0007_enqueue,
        (void(*) (void*)) Terminate0007_enqueue,
        (void(*) (void*)) Apply0007_enqueue, 0);
}
```

And then set a breakpoint in that method. For example in msdev, the breakpoint for the `internal/enqueue` primitive can be set using the Breakpoints dialog:



If you can add conditions to your breakpoints and the conditions can call functions, then you can break on the firing of a particular instance. For example, the breakpoint above can have the condition added `lastClosureIndex() == 9` or `lastClosureIndex() == 14` to distinguish between the two instances of the `internal/enqueue` box. (However, in msdev it should be noted that setting conditions this way failed. It may not be possible in msdev to use function calls in the conditions.)

Case 1179: Launch Package Compression

A launch package is a directory containing a deliverable Gedae application. Launch packages can reside on disk or be burnt into an EPROM. In either case, it is desirable to make the launch package as small as possible. The goal of this ongoing task is to reduce the size of launch packages significantly. There are two different ways to generate a launch package. The launch package creation dialog allows the user to choose between "Create Application Description in Command Program" and "Create Application Description on Disk". Work is being done in both areas to reduce launch package size as described in the subsections below.

Launch packages are generally proportional in size to the flattened graph from which they are generated. Large graphs are usually a result of using families to replicate primitives in the graph. Such replication creates redundancy in the launch package that can be exploited by an appropriate compression algorithm. Another way of reducing launch package size is to eliminate the storage of zero valued data. Both approaches are used.

Create Application Description in Command Program

When the "Create Application Description in Command Program" toggle is selected, the launch package control information is code-generated as data structures compiled into the command program. The file that is code-generated, `launch.c`, contains a complete description of the launch package control information. For graphs with large family sizes, reducing the size of this file will provide the most significant reduction in the launch package size.

There are several different blocks of data to compress in the file, but the basic technique is the same. The steps taken to compress the data are as follows:

1. The data structures in the memory map are sorted so that data structures of the same type are next to each other.
2. The data structures in the memory map are further sorted so that like structures associated with the same family are placed in contiguous memory and sorted by family index.
3. The memory map is created as a single memory block that can be code-generated directly into the `launch.c` file. This method of describing the memory map is different than the previous implementation where separate blocks were code generated for each structure or array of structures.
4. The block of memory created in step three is compressed by a new compression algorithm tuned to efficiently find family replication.
5. The compressed memory block is code-generated into the `launch.c` file along with any additional information required to decode the block.

This same compression technique is applied to two different areas in the `launch.c` file. Other areas also need to be compressed and these compressions will be implemented in subsequent releases.

Comparison of the `launch.c` file and `exec-host.exe` files created for the graph `demo/stap/rt_stap_hard_8_16` mapped to one processor show the following improvement:

Filename	Original	Gedae 4.5.1
		Compressed
<code>launch.c</code>	1.86Mbytes	0.72Mbytes
<code>exec-host.exe</code>	1.38Mbytes	1.11Mbytes

The goal of the compression algorithm is to make the `exec-host.exe` file as small as possible. While the areas that have been compressed are reduced by a factor of about 5, the remaining areas now dominate. This comparison will be continued in future releases.

Two parameters for controlling the launch package compression can be adjusted from the command line. These are called the "control-structure stride" and the "gen-launch stride". The default for these values is 200. A value of zero implies no compression. Larger numbers are more aggressive in looking for compressible areas – though larger may not always produce superior compression. These parameters can be set as:

```
gedae ... -cs_st 400 -gl_st 40 ...
```

In this example, the control-structure stride has been set to 400 and the gen-launch stride has been set to 40.

Create Application Description on Disk

The default method for creating launch packages is to create disk files describing the launch package control information. These files are read by the command program at runtime, and the information read is converted into the data structures needed. The most significant are the files named `proto-<partname>`. For example, the file created for partition default is named `proto-default`. The sizes of the protocol files that are created when an application description is created on disk were previously quite large due to an unnecessary storing of zero value data. Eliminating these zero values resulted in a `proto-default` file that was reduced in size from 28.5 Mbytes to 1.1Mbytes. This reduction in the size of the protocol file will make loading launch packages significantly faster.

Case 1190: Setting Options Parameter of `taskSpawn` in `evxworks BSP`

In the `solaris/evxworks BSP` target, executables are started using `taskSpawn`. Function `taskSpawn` has the prototype:

```
int taskSpawn
(
    char * name,          /* name of new task (stored at pStackBase) */
    int priority,        /* priority of new task */
    int options,         /* task option word */
    int stackSize,      /* size (bytes) of stack needed plus name */
    FUNCPTR entryPt,    /* entry point of new task */
    int arg1,           /* 1st of 10 req'd task args to pass to func */
    int arg2,
    int arg3,
    int arg4,
    int arg5,
    int arg6,
```

```
int    arg7,  
int    arg8,  
int    arg9,  
int    arg10  
)
```

The evxworks BSP allows the user to set both the priority and the stack size of the spawned task; however, there was no way to set the options parameter. This problem has been fixed. The options parameter can now either be set in the mapping table or it can be set in the embedded_config file. For example, the user can set the option in the mapping table to

```
-op 0x400
```

or equivalently

```
-op 1024
```

The complete list of settable processor parameters – including the priority and the stack size – can be viewed from the mapping table by selecting the processor and then selecting the Options->View Settable Parameters menu.

These same parameters that are set in the mapping table can be set as the defaults for a processor in the embedded_config file ProcDesc info section. For example:

```
Processor_Types: {  
  vxworks: {  
    Type: "evxworks"  
    Make_Params: "evxworks/runtime_make_info"  
    Info: "-op 0x408 -st 50000"  
    Memory_Desc: {}  
  }  
}
```

sets the default option field to taskSpawn to 0x408 and the default stack size to 50000 bytes.

2 Bugs Fixed

The following is a list of the bugs that have been fixed.

Case 1100: Change Embedded Build to Not Use gmake

This feature added in Gedae 4.5 was not completed correctly. As a result it was no longer possible to create launch packages. This problem has been fixed.

Case 1166: \$GEDAE/tools/emb_makefiles/printEmbEnv

Some versions of perl were unable to read the perl script found in file:

```
$GEDAE/tools/emb_makefiles/printEmbEnv
```

This problem has been fixed by making all invocations of perl from Gedae use the version of perl supplied with Gedae.

Case 1167: Print Statements Slow Scheduling

Print statements of the form:

```
markForwardFeedbackLoop YYY_42  
markReverseFeedbackLoop ZZZ_1  
markReverseFeedbackLoop ZZZ_2  
...  
markReverseFeedbackLoop ZZZ_39  
unmarkBoxElement AAA_1  
unmarkBoxElement AAA_2
```

were causing slow development time scheduling. The print statements were removed.

Case 1181: Double Arrays on Solaris

A bus error occurred on Solaris processors when a double array was connected from an Eval box output to an Eval box input. The alignment problem causing this bus error was fixed.

Case 1188: Inconsistent Prototyping of embCalloc

The prototypes for embCalloc were inconsistent in different .h files supplied with Gedae. The prototype is now consistently defined as:

```
void *embCalloc(char *mem_type, int elements, int size)
```

Case 1189 Minor Problems Causing Seg Faults

Four minor problems that caused the Gedae Development environment to segfault were fixed.

Case 1192: Transfer Parameters Are Set Wrong When Multiple Destinations Are on the Same Processor

Transfer parameters are unreliably set when a box output is mapped to one processor and is connected to several box inputs all mapped to the same destination processor. This problem was fixed.

3 Known Bugs

The Known Bugs are the same as those reported for Gedae 4.5. No new bugs have been added to this list.

Case 1024: Accommodate Different Exceed Versions in `initGEDAE`

In order for Gedae to work with Exceed/XDK 9.0, `makeGEDAE` had to be modified. This could be accomplished by a question or parameter to `initGEDAE`.

Currently users must view the FAQ for 7.1 changes
http://www.gedae.com/SUPPORT/FAQ/exceed_7_1.html
and then remove `xlibcon` from `makeGEDAE` and `ent` files mentioned in the FAQ.

Case 1026: Broadcast Transfer Mechanisms

Currently all Gedae data communication is point to point; however, many target processors support efficient broadcast mechanisms. The goal of this task is to extend the Gedae BSP API to allow broadcast data transfers to be part of the BSP. A second goal is to fix the DSA mechanisms that cannot be implemented when there is fanout (see Case 1149). Rather than fixing that problem directly, a broadcast DSA capability will be implemented.

Case 1056: FGU of Hierarchical Typedef Boxes

FGU does not transfer hierarchical typedef boxes correctly. The typedef used to define the input of the box is set to the old directory rather than the new.

Case 1057: Graph Stalls

A rare condition can cause a graph to stall (or segfault) when the controlled static schedule is partitioned to two processors in the following form:

A->B->A

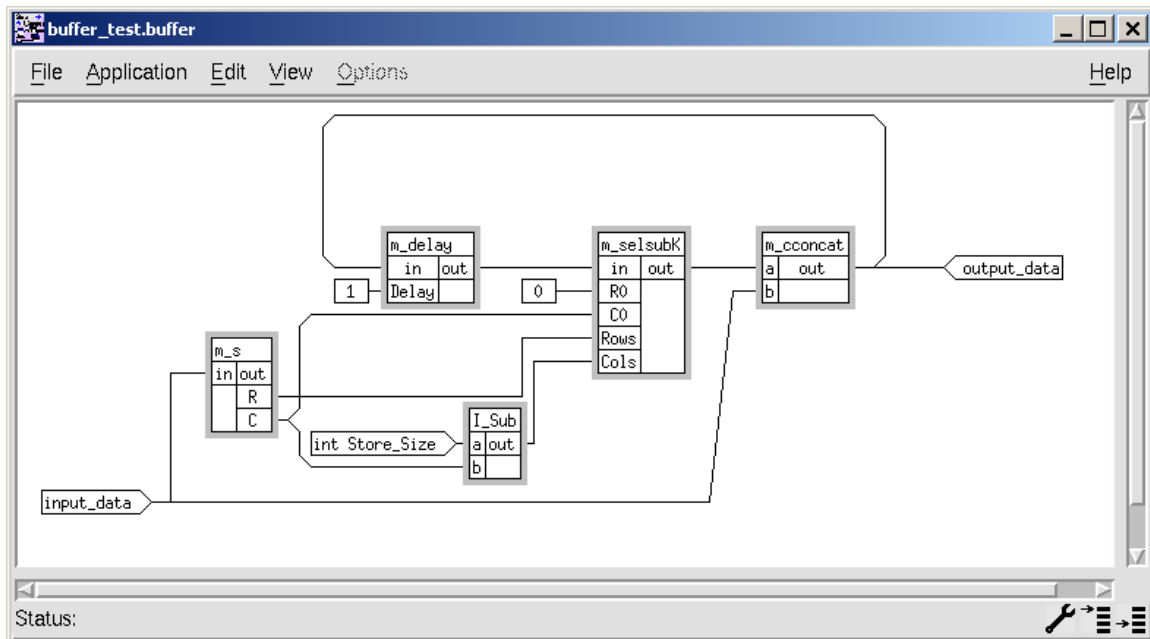
The problem scenario is that the schedule is partitioned into three parts, with the first and last parts mapped to the same processor. Usually Gedae puts the parts mapped to processor A in the same static schedule; however, to allow efficient pipelining, Gedae splits the two parts mapped to processor A into two different static schedules. They are

numbered n.1 and n.2 (for example 2.1 and 2.2). To see if any schedules have been broken into two parts, the user can pop up the Schedule Info Dialog and see if any of the schedule names contain a decimal point.

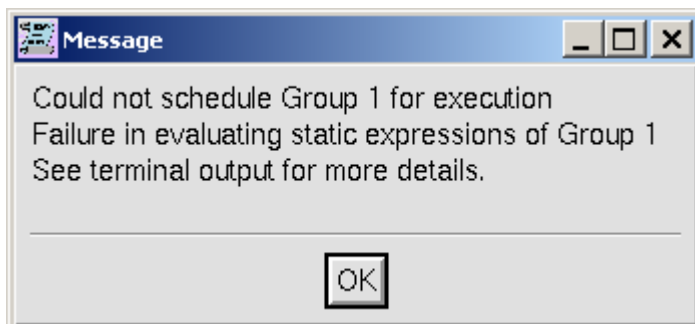
The decimal point in the schedule name does not necessarily indicate a problem. The problem only occurs when the data source driving the processing is faster than the graph, causing the control message queue to back up and overflow. The condition is rare because the problem only happens when the graph is not keeping up with the input data rates.

Case 1071: Dimensions Not Set Correctly

The dimensions are not set correctly in the graph below due to an interaction between the output parameter from the `m_s` box and the propagation of the dimensions around the feedback loop.



The error manifests itself as:



The workaround is to directly set the `m_selSubK` input parameters from graph parameters rather than from the `m_s` output.

Case 1078: Gedae Overwrites Protected Files

Gedae overwrites files that are protected as a result of being checked out of a CMS. This overwriting causes a problem because it defeats the CMS system.

Case 1108: Setting Default Subschedule Queue Capacities Correctly

The queue policy and capacity of a queue feeding a subschedule may not be set correctly.

Case 1122: Embedded Build Can Require a `makeGEDAE CLEAN`

If an application is repartitioned, then the target executables don't get relinked. The problem is that all the `.o` files are older than the targets, and the fact that there is a new link line does not force the target library and target executables to rebuild.

Case 1127: Static Schedule Gain Setting Algorithm Propagates Through Dynamic Queues

If there is a dynamic queue internal to a static schedule, then the gain setting algorithm propagates the gain through the dynamic boundary. Since the decimate/interpolate rates set for such boundaries are not necessarily a reflection of the actual data flow gain, these rates can lead to a report of a gain error in the graph.

Case 1139: Unterminated Comments

Unterminated comments cause the Gedae parser to segfault.

Case 1140: Parser Problem

The Gedae parser does not handle an odd number of quotes (") well.

Case 1141: Problem with Variable Vectors and Delay

If a dynamic variable vector input is preceded by a `vv_delay`, then the graph segfaults during scheduling.

Case 1142: Arrays of Strings Not Allowed

Gedae currently allows string array graph parameters to be declared as:

```
const string X[] = {"hello", "world"}
```

or

```
string X[i] = [i]Y
```

where `Y` is a family of strings.

In either case, the values so declared are not correctly set, and therefore, should be considered illegal.

Case 1143: Inconsistent Data Type Declarations

Gedae aborts when the same stream type is multiply defined. For example, suppose two primitives define data types with the same name but different definitions. If the primitives use these types in their `Input`, `Local` or `Output` sections, then Gedae aborts.

Case 1144: Function `appFree` Memory Leak

A command program running on VxWorks does not free all the resources allocated (memory, sockets, etc). The `appFree` function must release everything allocated. Gedae should automatically generate a call to `appFree` for the standard `exec-host` command program.

Case 1145: External Code Does Not Recompile

`Make` is not called after a successful run, so code changes to code listed in the `Personal_Emb_Obj_List` do not get recompiled. To force the recompile, it is currently necessary to change something from the Gedae GUI - like saving a primitive or toggling the Group "Run On Embedded" toggle off and on.

Case 1146: Large Graphs Fail to Display on Flattened Graph

If a graph is too large, then it cannot be displayed on the flattened graph. That is, if the flattened width or height exceeds the allowable pixmap width or height.

Case 1147: Primitive Cannot Recompile

If a primitive `Input`, `Output` or `Local` section is modified at runtime, then Gedae segfaults when the primitive is recompiled, and the graph is rerun. The user must currently exit Gedae after a primitive `Input`, `Output` or `Local` section has been modified.

Case 1148: VxWorks `embWallclock` Function Misses Wrap

When collecting trace information, the `embWallclock` function timer can wrap without being detected. This failure to detect the timer wrap causes VxWorks processor timelines to appear compressed.

Case 1149: DSA with FanOut Does Not Work for Some BSPs

If one box output fans out to several boxes mapped to several different processors, then the DSA communication mechanism does not work correctly for Mercury and Sky BSPs.

Case 1150: Trace Table Saved on NT Not Readable on Solaris

Files saved from NT are byte reversed from what is expected on Solaris. Files saved on a big-endian platform cannot be read by Gedae from a little-endian platform.

Case 1151: FFT Primitives Only Work with Power of 2 Sized Vectors

The FFT boxes do not support non-power-of-2 lengths; however, the comments make no mention of this fact. If these boxes only support a power-of-2, then it would be useful to have a separate set of boxes that support a non-power-of-2.

Case 1152: Queues Don't Auto-resize When Opening Group Settings

If a queue needs to be autoresized and the autoresize option is turned on in the group-setting file, then the autoresize does not occur. A workaround for this problem is to toggle the Group Control Dialog Autoresize toggle off, and then on, and then resave the group settings.

Case 1153: Cannot Put Parentheses Inside Single Quotes

A line like `c = '{'` inside a primitive causes an error during primitive parsing. The parser will count the `'}` as a bracket and fail to find a matching closing bracket. Because brackets inside double quotes are ignored, the expression `"{"` will not cause a problem.

Case 1154: Set Partition by Equation Error

When setting a partition by equation on a subgraph family, and if any of the contained boxes are also families, then the `$1` variable applies to the deepest family member giving the wrong results.

Case 1155: Constants Propagated Through typedef Boxes

Constants propagated through typedef boxes cannot be used for instantiation.

Case 1156: Partially Connected State Variable

State variables should not have to be connected to a primitive in every branch of an exclusive output. Currently, they must be connected when the branches are distributed.

Case 1157: Segmented Schedules with `nondet` Inputs

If a segmented static schedule has only `nondet` input queues, then it will not respond to segment ends correctly. Such schedules should be disallowed at development time. It is possible that in some situations such schedules should be legal, and this possibility is also being investigated.

Case 1158: Primitives with EndOfSegment and No Apply

Primitives that have an `EndOfSegment` method but don't have an `Apply` method do not get included; therefore, the `EndOfSegment` method does not run. A workaround for this problem is to include an empty `Apply` method in the primitive.

Case 1159: Stream Box with push in Hostless Launch Package

If a stream box contains a call to `push` and it is made part of a hostless launch package, then the launch package will fail to compile, as the code for the `push` is not included in the standalone library.

Case 1161: avail on Variable Vectors is Unreliable

On variable vectors, the `avail` function only returns the number of tokens available on the vector and not on the variable part. Usually these are the same, but when the variable vector is coming from another processor over a dynamic queue they may be updated at different times and may be different. As a result, `avail` may return a number too large. When the data is processed, the variable part may not actually be ready, causing a segfault. A workaround is to check the `queues_ready` flag after the call to `amount` and only continue with execution if the `queues_ready` flag is non-zero.