Chapter 18. CGC Domain

Authors: *Soonhoi Ha*

Other Contributors: *Mudit Goel*

In this chapter, we will explain the current implementation of C code generation domain. The source code can be found `$PTOLEMY/src/domains/cgc/kernel` directory. We follow the general framework for code generation defined in CG kernel directory.

In the CGC domain, the resource we have is the name space. We have to avoid name conflicts by guaranteeing unique names for different variables. The most complicated task is to determine the dimension, or buffer size, of each variable, and the method how to access them; static buffering, linear indexing, or modulo addressing.

We use the CGC domain to test new functionalities in code generation: buffer embedding for example. We have tested some simple demos to verify the design.

18.1 Buffer Allocation

In the CGC domain, we allocate one buffer for each connection in principle. We have to determine the required size of buffers first. If a porthole is *embedded* , and the buffer size requirement is equal to the sample rate of the embedded port, we do NOT allocate a buffer on that connection. We will use static buffering for all *embedded* and *embedding* portholes. If the buffer requirement of an embedded(or embedding) porthole is not equal to the sample rate of the porthole, we actually need to have two buffers on that connection and copy data between these buffers. In this case, we splice a Copy star on the arc and schedule the Copy star appropriately to generate code for copying data. After inserting the Copy star, we will end up with one buffer per connection. Another cause of copy requirement is type conversion from complex to float/int or from float/int to complex. Then, we splice a type-conversion star on the arc.

Class CGCTarget redefines the following protected method for buffer allocation .

```
int allocateMemory();
```

In this method, we first merge cascaded forks into a single fork whose input keeps the list of all fork destinations. We will allocate only one buffer for each fork. All fork destinations will refer to the same fork input buffer. Then this method does the following tasks:

1. Determine the buffer requirements for all portholes.
2. Splice Copy stars or type conversion stars if necessary.
3. Set the buffer type for each output porthole: either OWNER or EMBEDDED. If the output porthole is embedded, or the corresponding input porthole is embedded, it is called EMBEDDED. Otherwise, it is OWNER. The buffer type of an output is determined using the following public method of CGCPortHole class:

```
void setBufferType();
```

4. We assign unique names for buffers.

5. We initialize the offset pointer for each porthole which is associated with a buffer of size greater than 0 (`initOffset` method of CGCPortHole class). This offset pointer indicates from which offset of the buffer the porthole starts reading or writing samples.

`int initOffset();`

This is a public method of CGCPortHole class to initialize the offset pointer. If there are delays, or initial samples, on the arc, these samples are placed at the end of the buffer. The offset pointer of a porthole indicates the location of the last sample the next firing of its parent star will produce or consume. It is compatible with the SDF simulation domain: `$ref(porthole,num)` in CGC stars is now equivalent to `porthole%num` in SDF stars. We can set the offset pointer of an output porthole manually by the following public method of CGCPortHole class.

`void setOffset(int` *`v`*`);`

Now, we will explain steps (1), (2), and (4) in more detail.

18.1.1 Buffer requirement

To determine the buffer requirements of portholes, we traverse portholes of all stars, and call `finalBufSize` method of CGCPortHole class.

`void finalBufSize(int` *`statBuf`*`);`

This is a public method of CGCPortHole class to determine the buffer size for this porthole. The argument indicates whether we try to use static buffering or not. We allocate one buffer for each connection. Therefore, we do nothing if this porthole is an input porthole. If this porthole is disconnected, we set the buffer size equal to the number of samples produced for each firing. If it lies at wormhole boundary, we use `localBufSize` method of CGPortHole class to determine the size of buffer and return. Otherwise, we do the following:

1. We can manually assign the buffer size by calling `requestBufSize` for an output porthole of interest in the setup stage of a star:

`void requestBufSize(int` *`sz`*`);`

This method sets the buffer size manually. The argument size should not be smaller than the minimum size determined by the scheduler. The minimum size determined by the scheduler is the sum of maximum number of samples accumulated on the arc during the schedule and the number of old samples to be access from the destination port. If *`sz`* is smaller than this minimum value, we generate a warning message and give up manual allocation.

2. We set the initial buffer size by calling `localBufSize` method of CGPortHole class. If argument *`statBuf`* = 1, we set the buffer size as a smallest multiple of the sample rate of this porthole, which is not less than the initial buffer size. By doing this, we increase the chance of using linear buffering. We also set the waste factor in CGCGeodesic class to a huge number by calling the following public method in

CGCGeodesic class:

```
void preferLinearBuf(int i)
```

The waste factor set by the above method can be obtained by the following redefined protected method of the CGCGeodesic class.

```
double wasteFactor() const;
```

3. We set two flags for this porthole to indicate we can use static buffering and/or linear buffering: `hasStaticBuf` and `asLinearBuf`. These two flags are all private. If static buffering flag is set, we use direct addressing in the generated code to access the buffer. If linear buffering flag is set, we will use indirect addressing and no modulo addressing will be required. Otherwise, we will use indirect addressing and modulo addressing in the generated code to access the allocated buffer. Initially both flags are set TRUE. If this porthole needs to access past samples, we reset both flags to FALSE. When the argument *`statBuf`* is given 0, we give up static buffering in case the buffer size determined in (2) is greater than the sample rate of this porthole. Note that if a loop scheduler is used, *`statBuf`* becomes 0 and some possibilities of static buffering are sacrificed as the cost of code compaction. The following method is called to adjust the flags further.

```
void setFlags();
```

Is a protected member of CGCPortHole class. If the final buffer size is not a multiple of the sample rate, we reset `asLinearBuf` flag to 0. We have to use modulo addressing in the generated code. If the product of the sample rate and the repetition count of its parent star is not a multiple of the final buffer size, we give up static buffering, setting `hasStaticBuf` to 0. If an output porthole is embedded or embedding, we set both flags TRUE since we enforce static buffering.

4. As the final step, we set the flags for destination portholes. If this porthole is connected to a fork input, all fork destinations will be the destination portholes of this porthole. We first check whether *`statBuf`* argument is 0 and the buffer size is greater than the sample rate of the porthole. And, we call `setFlags` method for that porthole. If the porthole needs to access past samples, or the number of initial samples on the connection is not a multiple of the sample rate, we give up linear buffering.

The final buffer size can be obtained by the following two public methods of CGCPortHole class.

```
int maxBufReq() const;
int bufSize() const { return maxBufReq(); }
```

The above methods return the final buffer size associated with this porthole. If it is a fork destination, it returns the size of the fork input buffer. If the porthole has switched its Geodesic , it returns the size of buffer associated with the switched Geodesic.

The flags for static buffering and linear buffering can be obtained by the following public methods of CGCPortHole class:

```
int linearBuf() const;
int staticBuf() const;
```

We give up static buffering for a CGPortHole by calling the following public method of CGCPortHole class.

```
void giveUpStatic();
```

18.1.2 Splice stars

After buffer requirements for all portholes are determined, we can detect the arcs which can not have only one buffer. For instance, if we need to convert data types from complex to float/int or vice versa automatically, we need two buffers on the arc: one for complex variables and the other for float/int variables. This copying operation is required since C language does not provide built-in "complex" type variable. Therefore, we define "complex" type data in the generated code as follows;

```
static char* complexDecl =
"\back n#if !defined(COMPLEX_DATA)\back n#define COMPLEX_DATA 1"
"\back n typedef struct complex_data { double real; double imag; }
complex; \back n"
"#endif\back n";
```

Another case is when an embedded or embedding porthole requires a buffer whose size is greater than the sample rate of the porthole. Recall that an embedded or embedding porthole will assume static buffering for each execution when we generate code for that porthole. If the buffer size is larger than the sample rate, we may not use static buffering. We need two buffers for the embedded or embedding porthole.

Rather than assigning two buffers on an arc and letting the target generating code to copy data between these two buffers, we splice a star on the arc. The spliced star will separate two buffers on one arc into one buffer on its input and the other buffer on its output arc. When this spliced star is scheduled before the destination star of after the source star, it will generate code to copy data from the input buffer to the output buffer.

Stars are spliced in the following protected method of CGCTarget class.

```
void addSpliceStars();
```

This method traverses all portholes of stars in the galaxy.

When we splice a star at an input porthole (destination porthole), we initialize the spliced star and set the target pointer. A spliced star should have one "input" and one "output". We set the sample rate of these portholes equal to the sample rate of the input porthole. The buffer size of the input porthole of the spliced star is determined by the original source porthole. The buffer size of the output porthole is set the sample rate of the input porthole. And, we check whether static or linear buffering can be used for the portholes. The input porthole of a spliced Copy star gives up static buffering while the output porthole of the spliced Copy star and the original destination porthole can use static and linear buffering. In case we spliced a type conversion star, we need to change the type of the original source porthole.

We splice a Copy star at the output (source porthole) when the output is an embedded or embedding porthole and the buffer size is larger than the sample rate of the output porthole. We initialize the spliced star and set the target pointer. The sample rate of the input and output porthole of the spliced Copy star is equal to the sample rate of the output porthole. The buffer size of the output porthole of the spliced star is set to the buffer size of the arc. We give up

static buffering for this output porthole. On the other hand, we change the buffer size of the source porthole to the sample rate of the porthole.

We need to pay special attention to Collect (or Spread) stars. A Collect (or Spread) star is not a regular SDF star so that it is not scheduled when all input data are available. Actually, we do not execute the spliced Collect (or Spread) stars. But, the output porthole of a Collect (or Spread) star is an embedding (or embedded) porthole. And its buffer size can be larger than the sample rate of the porthole. In this case, we splice a Copy star at the destination porthole, not at the source porthole. We schedule this Copy star before the destination star. The sample rate of portholes of the spliced Copy star is equal to the sample rate of the destination porthole. The output buffer size of the spliced star is set the the buffer size of the arc while the input buffer size now becomes the sample rate of the source porthole. The trickest part here is to determine the offset pointers. We copy data when the destination porthole requires it. Therefore, the offset pointers of the input porthole and the output porthole of the spliced Copy star depends on the initial delay on the arc. We manually set the offset by `setOffset` method of CGCPortHole class.

There is another case we need data copying between two buffers: when two embedded portholes are connected together. Suppose, an output porthole of a Spread star is connected to an input porthole of a Collect star. Since the output porthole of a Spread star is embedded to the input buffer and the input porthole of a Collect star is embedded to the output buffer, we need to copy data from the input buffer of the Spread star to the output buffer of the Collect star. Since we do not schedule neither Spread nor Collect star, we may not splice a Copy star either at the source porthole not at the destination porthole. Therefore, we leave it as a special case so that we generate code to copy data between two buffers in `moveDataBetweenShared` method of CGCStar class after executing the star connected to the input porthole of the Spread star. So, we do not splice star when two embedded portholes are connected together.

```
void moveDataBetweenShared();
```

This is a protected method of CGCStar class. This method is called inside `runIt` method after generating code for a star. If the star is connected to an embedding porthole of a star of which an embedded output porthole is connected to an embedded porthole. Since we meet the case when two embedded portholes are connected, we generate code for copying data between two embedding buffers.

Scheduling spliced stars

When we splice a star at the input port of a star, we want to schedule the spliced star before the star. On the other hand, we want to schedule the spliced star after a star if we splice a star at the output porthole of the star. When we splice stars, we are already given the schedule. Therefore, we need to insert spliced stars into the schedule. An intuitive approach is to insert them into the schedule list.

Currently, we use a simpler method. We use the fact that the spliced star and the star connected to the spliced star can be regarded as a cluster and schedule of that cluster is well known. Our idea is to actually execute the cluster when we execute a star if the star is connected to spliced stars. CGCStar class has a private member to keep the list of stars: `spliceClust`. Initially, the star itself is inserted to the list. If we splice a star at the input porthole, we prepend the spliced star to the list. If we splice a star at the output porthole, we append the

spliced star to the list. And, we redefine `run` method.

```
int run();
```

If there are spliced stars, or the list size is greater than 1, we traverse the list and execute `runIt` method for each star. Otherwise, we execute `runIt` method.

```
int runIt();
```

It is a protected method of CGCStar class to generate main code for this star. If generates a comment regarding this star and main code. It updates offset pointers of the star. Finally, it calls `moveDataBetweenShared` method to generate code to copy data between two embedding portholes if necessary.

18.1.3 Buffer naming

One major task for resource assignment in the CGC domain is to give a unique name for each variable. In the setup stage of the CGCTarget, we assign a unique index value to each star starting from 1 to the number of stars in the galaxy. The CGCTarget has two protected members to give a unique index for galaxy.

```
int galId;
int curId;
```

The second member is used to give unique indices for galaxies while the first member indicates the index of the current galaxy.

Now, the CGCTarget can generate a unique name for each variable, portholes and states, by the following protected method.

```
StringList sanitizedFullName(const NamedObj& b) const;
```

In this method, the argument object is a porthole or a state of a star. We prefix ’g’ followed by the galaxy index, followed by "_", followed by the name of the star, followed by another ’_’, followed by the star index, followed by yet another ’_’ to the name of the object. For example, if star A has a state xx and the star index is 2 and the galaxy index is 1, the name of the state becomes "g1_A_2_xx".

```
StringList correctName(const NamedObj& b);
```

Is a public version of `sanitizedFullName` method.

Now, we are ready to generate unique names for portholes.

```
void setGeoName(char* name);
```

Is a public method of CGCPortHole class. If this porthole is disconnected and no Geodesic is assigned, we store the name in the porthole. Otherwise, we store the name in the Geodesic by calling the following public method of CGCGeodesic class.

```
void setBufName(char* name);
```

The buffer name of a porthole can be obtained by the following public method of CGCPortHole class.

```
const char* getGeoName() const;
```

This method returns the buffer name stored in this object if it is disconnected, or call `getBufName` method of CGCGeodesic class. If it is a fork destination, it returns the name of

the fork input buffer.

18.2 Data structure for galaxy and stars

In the global declaration section of the generated code, we declare data structures for stars. At early design stage of CGC domain, we use `struct` construct of C language to declare the data structure of the program. This way, we could assign unique memory locations to variables very easily. But, the length of a variable gets large as the hierarchy of the graph grows. Furthermore, we reduce significant amount of compiler optimization possibility. Therefore, we invented a scheme to generate unique symbols for variables (`sanitizedFullName` of CGCTarget class) without using "struct" construct.

`virtual void galDataStruct(Galaxy& galaxy, int level = 0);`
`virtual void starDataStruct(CGCStar* block, int level = 0);`

The above methods are protected methods of CGCTarget class to be called in `frameCode` method to declare data structures of galaxy and stars. The second argument of both methods indicates the depth of hierarchy which the first argument block resides in, thus advising the amount of indents in the generated code. By default, it is set 0. The first method calls the second method for each component star if it is not a Fork star. We do not generate code nor declare data structure for Fork stars.

The data structure for a star consists of four fields:

1. Comments to indicate that the following declarations corresponds to what star: `sectionComment` method.

`StringList sectionComment(const char* string);`

This is a protected method of CGCTarget class to generate a comment line, *string* in the generated code.

2. Declare buffers associated with portholes. We do not declare input portholes. If an output porthole is EMBEDDED, we declare a pointer to the embedding buffer, by prepending '*' in front of the buffer name. Otherwise, it declare a regular buffer.

3. Declare index pointers to the buffer if static buffering is not used and the size of buffer is greater than 1 . Portholes will use these index pointers to locate the buffer position. For a regular buffer, we declare an index pointer, named after the buffer name appended by "_ix". The name of index porthole is given by `offsetName` method of CGCTarget class.

`StringList offsetName(const CGCPortHole* p);`

This is a public method to assign an index pointer to the argument porthole. It appends '_' followed by "ix" at the end of the porthole name, by calling the following public method of CGCTarget class:

`StringList appendedName(const NamedObj& p, const char* add);`

This method is used to append '_' followed by *add* to the name of the object *p.*

4. Finally, we declare referenced states. A State is called *referenced* only when we

use $ref macro for the state at most once. CGCStar class has the following members for referenced states:

```
StateList referencedStates;
void registerState(const char* name);
```

The first is a public member to store the list of referenced states in this Star. The second is a protected method to add the state with given name to the list of referenced states if not inserted.

We traverse the list of referenced states to declare variables. Unlike portholes, the size of a state variable is given. If the size of state is 1, we both declare and initialize the state. If the state is an array state, we both declare and initialize the state using array initialization unless the state is declared inside a function. If we declare an array state inside a function, we have to write explicit initialization code. Class CGCTarget has the following public method to tell whether we are working inside a function or not.

```
int makingFunc();
```

Returns TRUE if we are defining a function.

18.2.1 Buffer initialization

We initialize buffers and index pointers as follows.

1. If the buffer is EMBEDDED, we assign a pointer to the embedded buffer and set the pointer to the starting address of the embedding buffer, from which the buffer is embedded. If the size of the embedding buffer is 1, we assign the pointer of the embedding buffer.
2. For the regular buffer, we initialize with 0s in case the buffer size is greater than 1.
3. We initialize an index pointer of a buffer to the offset pointer of the porthole associated with that index pointer.

18.3 CGC code streams

Besides two code streams inherited from CGTarget class, `myCode` and `procedures`, CGCTarget class maintains 9 more code streams (all protected). These code streams will be stitched together to make the final code in `frameCode` method. There are two schemes to organize a code in general. One scheme would be to put code strings to a single CodeStream in order. For example, we put global declarations, main function declaration, initialization, and main loop into a single `myCode` stream in order. For single processor code generation, it would be feasible. For multiprocessor case, however, the parent target may add some extra code strings. Therefore, we assign different code streams to different section of code. On the other hand, if we have too many code streams, it would be arduous to remember all.

```
CodeStream globalDecls;
CodeStream galStruct;
CodeStream include;
```

These three code streams will be placed in the global scope of the final code. The galaxy declaration (`galStruct`) is separated from `globalDecls` because we need to put galaxy declaration inside a function if we want to define a function from a galaxy (for exam-

ple, recursion construct). A programmer can provide strings to `globalDecls` and `include` by using the following protected CGCStar methods in a star definition:

```
int addGlobal(const char* decl, const char* name = NULL);
int addInclude(const char* decl);
```

In the first method, we use *`decl`* strings as the name if the second argument is given NULL, to make a global declaration unique. The argument of the second method is the name of a file to be included, for example <stream.h> or "DataStruct.h".

```
CodeStream mainDecls;
CodeStream mainInit;
CodeStream commInit;
```

These three code streams will be placed in the main function before the main loop: declaration in the main function, initialization code, and initialization code for communication stars. We separated `commInit` from `mainInit` since communication stars are inserted by the parent multiprocessor target. A programmer can provide strings to the first two code streams by using the following protected CGCStar methods.

```
int addDeclaration(const char* decl, const char* name = NULL);
int addMainInit(const char* decl, const char* name = NULL);
```

The first method uses *`decl`* string as the name of the string if *`name`* is given NULL.

```
CodeStream wormIn;
CodeStream wormOut;
CodeStream mainClose;
```

The first two streams contain code sections to support wormhole interface to the host machine. They will be placed at the beginning of the main loop and at the end of the main loop. The last code stream will be placed after the main loop in the main function.

Recall that using `addCode` method defined in CGStar class, we can put code strings to any code stream .

These nine code streams are initialized by the following protected method of CGCTarget class.:

```
virtual void initCodeStrings();
```

Note that code streams are not initialized in `setup` method of the target since the parent target may put some code before calling the `setup` method of the target. We initialize code streams after we stitch them together and copy the final code in `myCode` stream in `frameCode` method. We do not initialize `myCode` stream in the above method.

```
void frameCode();
```

This method put all code streams together and copy the resulting code to `myCode` stream.

18.4 Other CGCPortHole members

CGCPortHole is derived from CGPortHole class. It has a constructor with no argument. In the constructor, we initialize the default properties of a CGCPortHole: static buffering and linear buffering flags are set TRUE, buffer size is set to 1. These properties are also initialized in `initialize` method. In the destructor, it deallocates the name of the buffer if stored in this

class (when this porthole is disconnected). All members described in this section are public.

```
CGCPortHole* getForkSrc();
const CGCPortHole* getForkSrc() const;
```
These methods return the fork input porthole (`forkSrc`) if this porthole is a fork destination. The second method is the *const* version of the first method.

```
CGCPortHole* realFarPort();
const CGCPortHole* realFarPort() const;
```
These method return the far side porthole. If the far side porthole is a fork destination, they return the far side porthole of the fork input, thus bypassing fork stars. The second is the *const* version of the first method.

```
CGCGeodesic& geo();
const CGCGeodesic& geo() const;
```
Return the geodesic connected to this PortHole, type cast. The second is the `const` version of the first method.

```
Geodesic* allocateGeodesic();
```
Allocates a CGCGeodesic.

```
void setupForkDests();
```
If this method is called for a fork input porthole, make a complete list of `forkDests` considering all cascaded forks.

```
int inBufSize() const;
```
This method returns the `bufferSize` of this porthole.

CGCPortHole has an iterator called `ForkDestIter.` It returns fork destinations one at a time. The return type is CGCPortHole.

The derived classes of CGCPortHole in the CGC domain are `InCGCPort, OutCGCPort, MultiCGCPort, MultiInCGCPort,` and `MultiOutCGCPort.`

18.5 Other CGCStar members

Class CGCStar is derived from CGStar class. It has a constructor with no argument. CGCTarget class is a friend class. It has a method to return the domain it lies in (`domain`) and a method for class identification (`isA`). In `initialize` method, we initialize `referencedStates` list. All other members described in this section are all protected.

```
CGCTarget* targ();
```
Returns the target pointer, type cast to CGCTarget.

```
StringList expandRef(const char* name);
StringList expandRef(const char* name, const char* offset);
```
The above methods resolve macro $ref. The *name* argument is a state name or a porthole name. If it is a state name, we put the state in the `referencedStates` list. In the second method, the second argument is the offset of the first argument (state or porthole). It can be a numeral, an IntState name, or a string. If it is an IntState, the current value of the state is

taken.

There are various ways to referring to a porthole. If the buffer size is 1, we use the buffer name or the pointer version depending on the type, EMBEDDED or OWNER. If the buffer size is larger than 1, we use direct addressing if static buffering is used. If static buffering can not be used, we use indirect addressing. The following method generates indirect addressing:

`virtual StringList getActualRef(CGCPortHole* p, const char* ix);`

This method generates an indirect addressing for the argument porthole *p* with offset *ix.* If we may not use linear addressing, we generate modulo addressing, in which the index is modulo the buffer size.

```
virtual int amISpreadCollect();
```

Returns TRUE or FALSE, based on whether this star is a Spread or a Collect star or not. We need to take special care for Spread and Collect stars.

18.6 Other CGCTarget members

CGCTarget is derived from HLLTarget class which is the base target class for high level language code generation. It has a constructor with three argument like its base target classes. In the constructor, we initialize code streams and put them into the `codeStringLists` by `addStream` method. It has `makeNew` method defined.

18.6.1 Other CGCTarget protected members

CGCTarget class has many states guiding the compilation procedure.

```
IntState doCompile;
```

If this state is set NO, we only generate code, not compiling the code.

```
StringState hostMachine;
StringState funcName;
StringState compileCommand;
StringState compileOptions;
StringState linkOptions;
```

The `hostMachine` state indicates where the generated code is compiled and run. If this state does not indicate the current host,, we will use remove shell command for compilation and execution. The `funcName` state is by default set "main". For multiprocessor code generation case, we may want to give different function name for the generated code. The next three states determines the compilation command:

`compileCommand` `compileOptions` fileName `linkOptions`

There are some other states defined in this class.

```
IntState staticBuffering;
```

If this state is set YES, we increase the `wasteFactor` of geodesics to use static buffering as much as possible, which is default.

```
StringState saveFileName;
```

We save the generated code in this file if the file name is given.

`StringArrayState resources;`
This state displays which resources this target has. By default, the CGCTarget has the standard I/O (*`STDIO`*) resource. If a derived target does not support the standard I/O, it should clear this state.

`int codeGenInit();`
This method generates initialization code: buffer initialization, and `initCode` method of all stars. Before generating initialization code, we switch the `myCode` pointer of stars to the `mainInit` code stream so that `addCode` method called inside the `initCode` method puts the string into the `mainInit` code stream.

`void compileRun(SDFScheduler* s);`
Before calling `compileRun` method of the SDFScheduler, which will call `run` method of stars in the scheduled order, we switch the `myCode` pointer of stars back to the `myCode` code stream of the target. After code generation, we switch the pointer of stars to the `mainClose` code stream for wrapup stage.

`int wormLoadCode();`
If the `doCompile` state is set NO, we just return TRUE, doing nothing. Otherwise, we compile and run the generated code. Return FALSE if any error occurs.

`StringList sectionComment(const char* s);`
This method makes a comment statement with the given string in C code.

`void wormInputCode(PortHole& p);`
`void wormOutputCode(PortHole& p);`
The above methods just print out comments. We haven't supported wormhole interface for CGC domain yet (Sorry!).

18.6.2 Other CGCTarget public members

`void setup();`
This method initialize `galId`, `curId` indices for unique symbol generation. It also generate indices for stars and portholes. Then, it calls `CGTarget :: setup` for normal setup procedure.

`void wrapup();`
This method displays the generated code stored in `myCode` stream. If the galaxy is not inside a wormhole, it calls `wormLoadCode` method to compile and run the code.

`int compileCode();`
This method compiles the generated code. The compile command is generated by the following method:

`virtual StringList compileLine(const char* fName);`
The argument for this method is the file name to be compiled. If the `hostMachine` does not indicate the local-host, we use remote shell.

`int runCode();`
This method runs the code. If the `hostMachine` is not the local-host, we use `rshSystem` function.

`void headerCode();`
Is redefined to generate a valid C comment with the target name.

`void beginIteration(int` *`repetitions`*`, int` *`depth`*`);`
`void endIteration(int` *`repetitions`*`, int` *`depth`*`);`
The first method generates the starting line of `while` loop (if *`repetitions`* is negative) or `for` loop (otherwise). After that it appends the `wormIn` code stream to the `myCode` stream before stars fill the loop body. The `wormIn` code stream is already filled. The second method close the loop. Just before closing the loop, it appends the `wormOut` code stream to the `myCode` at the end of the loop body.

`void setHostName(const char*` *`s`*`);`
`const char* hostName();`
The above methods set and get the `hostName` state.

`void writeCode(const char*` *`name`* = NULL`);`
If the argument is NULL, we use the galaxy name as the file name. This method saves the code to the file.

`void wantStaticBuffering();`
`int useStaticBuffering();`
These methods set and get the `staticBuffering` state.

`int incrementalAdd(CGStar*` *`s`*`, int` *`flag`* = 1`);`
We add the code for the argument star, *`s,`* during code generation step. If *`flag`* is 0, we add the main body of the star (`go` method only). Otherwise, we initialize the star, allocate memory, and generate initialization code, main body, and wrapup code.

`int insertGalaxyCode(Galaxy*` *`g`*`, SDFScheduler*` *`s`*`);`
We insert the code for the argument galaxy during code generation procedure. We give the unique index for the galaxy and set the indices of stars inside the galaxy. Then, it calls `CGTarget :: insertGalaxyCode` to generate code. After all, we declare the galaxy.

`void putStream(const char*` *`n`*`, CodeStream*` `cs`);
`CodeStream* removeStream(const char*` *`n`*`);`
The above methods put and remove a code stream named *`n.`*

18.7 Class CGCMultiTarget

Class CGCMultiTarget, derived from CGSharedBus class, models multiple Unix machines connected together via Ethernet. We use socket mechanism for interprocessor communication. Since the communication overhead is huge, we do not gain any speed up for small examples. Nonetheless, we can test and verify the procedure of multiprocessor code generation.

This class has five private states as follows.

```
IntState doCompile;
IntState doRun;
```
If these states are set YES, we compile and run the generated code.

```
StringState machineNames;
StringState nameSuffix;
```
We list the machine names separated by commas. If all machines names listed have the same suffix, we separate that suffix in the second state. For example, if `machineName` is "ohm" and `nameSuffix` is ".berkeley.edu", we mean machine named "ohm.berkeley.edu".

```
IntState portNumber;
```
To make socket connections, we assign port numbers that are available. For now, we set the starting port number with this state. We will increase this number by one every time we add a new connection. Therefore, it should be confirmed that these assigned port numbers should be available. If the Ptolemy program is assigned a port number in the future, then we will be able to let the system choose the available port number for each connection.

With the given list of machine names, we prepare a data structure called `MachineInfo` that pairs the machine name and internet address.

```
class MachineInfo {
      friend class CGCMultiTarget;
      const char* inetAdddrr;// internet address
      const char* nm;   // machine name
public:
      MachineInfo: inetAddr(0), nm(0) { }
}
```
This class has a constructor with three argument like its base classes. The destructor deallocates `MachineInfo` arrays if allocated. It has `makeNew` method and `isA` method redefined.

18.7.1 CGCMultiTarget protected members

```
void setup();
```
If the child targets are inherited, we also inherit the machine information. Otherwise, we set up the machine information. The number of processors and the number of machines names should be equal. Then, we call `CGMultiTarget::setup` for normal setup operation . At last, we set the `hostName` state of child targets with the machine names.

```
int wormLoadCode();
```
This method do nothing if `doCompile` state is NO. Otherwise, it compiles the code for all child targets (`compileCode`) . Then, it checks whether `doRun` state is YES or NO. If it is YES, we execute the code.

```
int sendWormData(PortHole& p);
int receiveWormData(PortHole& p);
int sendWormData();
int receiveWormData();
```

These method should be redefined in the future to support wormhole interface. Currently, they do same tasks with the base Target classes.

18.7.2 CGCMultiTarget public members

```
MachineInfo* getMachineInfo();
int* getPortNumber();
```
These methods return the current machine information and the next port number to be assigned.

```
DataFlowStar* createSend(int from, int to, int num);
DataFlowStar* createReceive(int from, int to, int num);
```
The above methods create CGCUnixSend and CGCUnixReceive stars for communication stars with TCP protocol.

```
void pairSendReceive(DataFlowStar* snd, DataFlowStar* rcv);
```
This method pairs a UnixSend star and a UnixReceive star to make a connection. We assign a port number to the connection. More important task is to generate function calls in the initialization code (`commInit` stream) of two child targets which these communication stars belong to. These functions will make a TCP connection between two child targets with the assigned port number. The UnixSend star will call `connect` function while the UnixReceive star will call `listen` function.

```
void setMachineAddr(CGStar* snd, CGStar* rcv);
```
This method informs the *snd* star about the internet address of the machine that the *rcv* star is scheduled on. The address is needed in `connect` function.

```
void signalCopy(int flag);
```
By giving a non-zero value as the argument, we indicate that the code will be duplicated in different set of processors so that we need to adjust the machine information of communication stars.

```
void prepCode(Profile* pf, int nP, int numChunk);
```
This method is also used to allow code replication into different set of targets.

```
DataFlowStar* createCollect();
DataFlowStar* createSpread();
```
These methods create CGCCollect and CGCSpread stars.

18.8 Status

Here are some points about the current status.

- Data Parallel star is not supported yet.
- Execution times of CGC stars are not well defined. They will vary processor to processor. We estimate them by looking at CG96 stars, or by counting the number of elementary operations. For heterogeneous multiprocessor case, we have to design a clean way of specifying these numbers.

- hSpread/Collect stars and buffer embedding are not supported in ASM domain. Since Spread/Collect stars are not supported, all ASM multiprocessor targets should set the `oneStarOneProc` state TRUE.

(4) The scheduling option, `adjustSchedule` is not implemented yet since the current graphical editor does not support "cont" function.

(5) Overlapped communication is not supported since we haven’t had any machine of that sort.

18.9 References

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

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

[3] S. Ha, Compile-time Scheduling of Dataflow Program Graphs with Dynamic Constructs, Ph.D. dissertation, U.C.Berkeley, 1992.

[4] J.L.Pino, S.Ha, E.A.Lee, J.T.Buck, "Software Synthesis for DSP Using Ptolemy," invited paper, Journal of VLSI Signal Processing, 1993.

[5] W.S.Wang, et al, "Assignment of Chain-Structured Tasks onto Chain-structured Distributed Systems," source unknown.