Chapter 14. CGC Domain

Authors: *Joseph T. Buck*
Soonhoi Ha
Edward A. Lee
Yu Kee Lim
Thomas M. Parks
José Luis Pino

Other Contributors: *Sunil Bhave*
Kennard White

14.1 Introduction

The CGC domain generates code for the `C` programming language. "Code Generation" on page 13-1 describes the features common to all code generation domains. The basic principles of writing code generation stars are explained in "Writing Code Generation Stars" on page 13-2.You will find explanations for codeblocks, macros, and attributes there. This chapter explains features specific to the CGC domain. Refer to the CGC domain chapter in the user's manual for an introduction to this domain.

14.2 Code Generation Methods

The `addCode` method is context sensitive so that it will 'do the right thing' when invoked from within the `initCode`, `go`, and `wrapup` methods of `CGCStar`. Refer to "Writing Code Generation Stars" on page 13-2 for documentation on `addCode`, including context sensitive actions and conditional code generation. There are several additional code-generation methods defined in the CGC domain. The `addInclude` method is used to generate `#include` *file* directives. The `addDeclaration` method is used to declare local variables within the main function. The `addGlobal` method is used to declare global variables outside the main function. As with `addCode`, these methods return `TRUE` if code was generated for the appropriate stream and `FALSE` otherwise. These methods are member functions of the `CGCStar` class.

`int addInclude (const char* file)`
: Generate the directive `#include file` in the `include` stream. The string *file* must include quotation marks (`"file"`) or angle brackets (`<file>`) around the name of the file to be included. Only one `#include file` directive will be generated for the file, even if `addInclude` is invoked multiple times with the same argument. Return `TRUE` if a new directive was generated.

`int addDeclaration (const char* text, const char* name = NULL)`
: Add text to the `mainDecls` stream. Use `name` as the identifying key for the code fragment if it is provided, otherwise use

text itself as the key. Code will be added to the stream only the first time that a particular key is used.

`int addGlobal (const char* text, const char* name = NULL)`
: Add text to the `globalDecls` stream. Use *name* as the identifying key for the code fragment if it is provided, otherwise use *text* itself as the key. Code will be added to the stream only the first time that a particular key is used.

`int addCompileOption (const char* text)`
: Add options to be used when compiling a C program. The options are collected in the `compileOptionsStream` stream.

`int addLinkOption (const char* text)`
: Add options to be used when linking a C program. The options are collected in the `linkOptionsStream` stream.

The following streams, which are used by the code generation methods just described, are defined as members of the `CGCTarget` class in addition to the streams defined by the `CGTarget` class.

`CodeStream include`
: Include directives are added to this stream by the `addInclude` method of `CGCStar`.

`CodeStream mainDecls`
: Local declarations for variables are added to this stream by the `addDeclaration` method of `CGCStar`.

`CodeStream globalDecls`
: Global declarations for variables and functions are added to this stream by the `addGlobal` method of `CGCStar`.

`CodeStream mainInit`
: Initialization code is added to this stream when the `addCode` method is invoked from within the `initCode` method.

`CodeStream mainClose`
: Code generated when the `addCode` method is invoked from within the `wrapup` method of stars is placed in this stream.

`CodeStream compileOptionsStream`
: Options to be passed to the C compiler which have been added using the `CGCStar::addCompileOption` method.

`CodeStream linkOptionsStream`
: Options to be passed to the linker which have been added using the `CGCStar::addLinkOption` method.

14.3 Buffer Embedding

Although many of the methods related to buffer embedding are actually implemented in the CG domain, only the CGC domain makes use of them at this time. The following function is defined as a method of the `CGPortHole` class.

`void embed (CGPortHole` *`port`*`, int` *`location`* `= -1)`
: Embed the buffer of *`port`* in the buffer of this porthole with offset *`location`*. The default *`location`* of -1 indicates that the offset is not yet determined.

For example, the following statements appear in the setup method of the `Switch` block. This causes the buffers of `trueOutput` and `falseOutput` to be embedded within the buffer of input.

```
input.embed(trueOutput,0);
input.embed(falseOutput,0);
```

14.4 Command-line Settable States

In the Ptolemy releases before Ptolemy0.6 the C programs generated by Ptolemy in the CGC domain did not take any command-line arguments. The state values of the various stars were set during compilation and thus hard-coded into the program. In order to change a state variable, the code had to be recompiled again (i.e. the universe had to be re-run within Ptolemy). This was time consuming, and it also placed unnecessary load on the machine. In Ptolemy0.6 and later, the CGC domain can generate C code that allow users to set the state values from the command-line, which allows runs with different parameters to be executed and compared quickly and easily.

Implementation

14.4.1 C code generated to support command line arguments

A sample of the additional code generated to support command-line arguments is shown below:

```
.
.
struct {
 double FOO;
 double BAR;
} arg_store = {1.0, 0.01,};

void set_arg_val(char *arg[]) {
 int i;
 for (i = 1; arg[i]; i++) {
 if ((!strcmp(arg[i], "-help")) \
      ||(!strcmp(arg[i], "-HELP")) \
      ||(!strcmp(arg[i], "-h"))) {
      printf("Settable states are :\n
             FOO\tdefault : 1.0\n
             BAR\tdefault : 0.01\n");
      exit(0);
 }
 if (!strcmp(arg[i], "-FOO")) {
      if (arg[i + 1])
       arg_store.FOO = atof(arg[i + 1]);
```

```
                continue;
          }
          if (!strcmp(arg[i], "-BAR")) {
               if (arg[i + 1])
                arg_store.BAR = atof(arg[i + 1]);
                continue;
          }
          }
}

/* main function */
       main(int argc, char *argv[]) {
       .
       .
        double value_11;
        double value_12;
       .
       .     // End of Declaration
        set_arg_val(argv);
       .     // Begin of Initialization
       .
        value_12 = arg_store.BAR;
        value_11 = arg_store.FOO;
       .     // Code
       .
       }
```

The default values (set by the "edit-parameters" command) are stored in the struct `arg_store`. The function `set_arg_val(argv)` scans the list of command-line arguments for `FOO` and `BAR` and sets the corresponding member in `arg_store`. It also builds up the help message (consist of the settable state names and their default values) to be printed when the program receives a '`-h`', '`-help`' or '`-HELP`' option. The state values are initialized to the corresponding `arg_store` members during the variable initialization stage. By doing this, a state will get its default value if it is not set on the command-line.

14.4.2 Changes in pigiRpc to support command line arguments

The pragma mechanism in the `Target` base class is used to specify the state that is to be made settable via command-line arguments as well as to store the name to be used on the command-line. In `CGCtarget`, these are stored as a character string in a `TextTable*` mappings (a pointer to a `HashTable` in which the data value and index are character strings) via the overloaded `pragma()` member functions.

A function, `isCmdArg(const State* state)`, is used to check whether '`state`' is to be set by a command-line argument. It calls `CGCTarget::pragma()` and scans through the `StringList` returned for the state's name. If found, the mapped name is return. Otherwise a null string is return.

Four new protected `CodeStream` are added to `CGCTarget` to store the additional codes:

`cmdargStruct` stores the struct members.

`cmdargStruct` stores the default values.

`setargFunc` stores the code segment in `set_arg_val()`.

`setargFuncHelp` stores the built-up help message.

Four new public member functions and four private ones are also added to `CGCStar` to generate the codes:

`cmdargStates()` calls `cmdargState()` to generate the members of `struct arg_store` using the mapped name returned by `isCmdArg()`.

`cmdargStatesInits()`
calls `cmdargStatesInit()` to generate the default values of the settable states.

`setargStates()` calls `setargState()` to generate the code segment to match the mapped name to the command-line options.

`setargStatesHelps()`
calls `setargStatesHelp()` to build up the help message.

These are called in the `CGCTarget::declareStar(CGCStar* star)` function after the global and main declarations have been generated. `CGCStar::initCodeState(const State* state)` is modified to generate the required initialization code if state is to be settable from the command-line.

In order for a `$val` state to be settable from the command-line, it has to be changed to a `reference state`. The `expandVal()` member function is overloaded in `CGCStar` to check if the "`name`" state is to be made settable from the command-line. If so, it is added to the list of referenced state so that it will be declared and initialized.

14.4.3 Limitations of command line arguments.

Currently, this implementation works only for scalar states with float or integer values. Extension to other types of state should be straight forward by simply adding the appropriate struct member declaration code in `CGCStar::cmdargState(const State* state)`. The `cmdargStatesInit()`, `setargState()`, `setargStatesHelp()` and `initCodeString()` member functions need to be modified accordingly to generate codes for the initialization, setting function, help message and assignment respectively of the new state variable.

Also, there is no provision to check for duplicate command-line names. If there are duplicates, Ptolemy will simply generate multiple `struct` members with the same name, and error will result in the generated code. To get around this, a new Tk interface could be written to specify and set the settable states and checking can be done at that level. Alternatively, it might be a better idea to use the `put()` method in `CodeStream` to add the `struct` member with its unique handle to the appropriate `CodeStream`. That way, there will not be duplicate `struct` members and state-variables could still reference the same member, so that two or more states could be set to the same value from a single argument on the command-line.

Another limitation is that the command-line capability only works for states of blocks at the top level. It will not work for states of `Galaxies` and `Universes`, and states that referenced other settable states. This could probably be solved by modifying the pragma mechanism to ensure that `pragmas` at the top level propagate all the way down to the contained blocks. By doing this, states will inherit `pragmas` from their parent `galaxies` so that these can be picked up by the `isCmdArg()` function, and the appropriate codes can be generated.

Certain states will affect the overall scheduling of the whole system, e.g. the *factor* of `upsampling` and `downsampling` stars, and changing these would mean that new code needs to be generated since the scheduling is hard-coded into the generated code. Thus these should not be allowed to take values from the command-line. A new attribute can be introduced to identify those states that should not be settable from the command-line. Warnings can then be generated if users attempt to specify these for command-line setting.

14.5 CGC Compile-time Speed

There are several areas that can affect the amount of time that it takes a CGC universe to compile, we discuss them below.

- Large sample rate changes and large delays can result in Ptolemy taking a very long time to generate C code. A symptom of this sort of problem is that the `pigiRpc` process will consume all the available swap and eventually crash. If you feel you need really large delays, James Lundblad suggests writing your own code in your stars that provides the same functionality as delays, but uses `malloc()` in the `initCode` section instead of the array that is created by the CGC Delay icon.
- C compiler optimizers do not work well with functions that have thousands of lines. The `main()` function of a CGC simulation may be too large for the peephole optimizer, causing the optimizer to take a long time to compile the file. Under `gcc`, you can pass the `-O0` option to turn off the optimizer.

14.6 BDF Stars

Because the class `CGCPortHole` is not derived from `BDFPortHole`, the `setBDFParams` method described in "BDF Domain" on page 8-1 is not available for code generation stars. Use the `setRelation` method of `DynDFPortHole` instead.

`void setRelation (DFRelation *relation*, DynDFPortHole* *assoc*)`

Specify the *`relation`* of this port with the associated porthole *`assoc`*. There are five possible values for relation:

`DF_NONE`	no relationship.
`DF_TRUE`	produces/consumes data only when *`assoc`* has a `TRUE` particle.
`DF_FALSE`	produces/consumes data only when *`assoc`* has a `FALSE` particle.
`DF_SAME`	signal is logically the same as *`assoc`*.
`DF_COMPLEMENT`	signal is the logical complement of *`assoc`*.

For example, the following statements describe the relationships among the portholes of the `Switch` block.

```
trueOutput.setRelation(DF TRUE, control);
```

```
falseOutput.setRelation(DF FALSE, control);
```

14.7 Tcl/Tk Stars

The `CGCTclTkTarget` class defines the `tkSetup` stream for Tcl/Tk stars.There is no special code generation function for this stream, so its name must be used with `addCode`. This is usually done from within the `initCode` method.

```
addCode(codeblock, "tkSetup");
```

The following functions, which are defined in the file `tkMain.c`, can be used within codeblocks of Tcl/Tk stars in the CGC domain.

`void errorReport (char* `*`message`*`)`

This functions creates a pop-up window containing *`message`*.

`void makeEntry (char* `*`window`*`, char* `*`name`*`, char* `*`desc`*`, char* `*`initValue`*`, Tcl CmdProc* `*`callback`*`)`

This function creates an entry box in a *`window`*. The *`name`* of the entry box must be unique (e.g. derived from the star name). The description of the entry box is desc. The initial value in the entry box is *`initValue`*.

A *`callback`* function is called whenever the user enters a `RET` in the box. The argument to the *`callback`* function will be the value that the user has put in the entry box. The return value of the *`callback`* function should be `TCL_OK`.

`void makeButton (char* `*`window, char* name`*`, char* `*`desc`*`, Tcl CmdProc* `*`callback`*`)`

This function creates a push button in a *`window`*. The *`name of`* the push button must be unique (e.g. derived from the star name). The description of the push button is *`desc`*.

A *`callback`* function is called whenever the user pushes the button. The return value of the *`callback`* function should be `TCL_OK`.

`void makeScale (char* `*`window`*`, char* `*`name`*`, char* `*`desc`*`, int `*`position`*`, Tcl CmdProc* `*`callback`*`)`

This function creates a scale (with slider) in a *`window`*. The name of the scale must be unique (e.g., derived from the star name). The description of the push button is desc. The initial position of the slider must be between `0` and `100`.

A *`callback`* function is called whenever the user moves the slider in the scale. The argument to the *`callback`* function will be the current position of the slider, which can range from `0` to `100`. The return value of the *`callback`* function should be `TCL_OK`.

`void displaySliderValue (char* `*`window`*`, char* `*`name`*`, char* `*`value`*`)`

This function displays a value associated with a scale's slider. The scale is identified by its name and the `window` it is in. This function must be called by the user of the slider. Only the first 6 characters of the value will be used.

14.8 Tycho Target

The CGC TychoTarget is an experimental target that provides a way to create CGC control panels that use the functionality in Tycho. A universe that uses TychoTarget must provide a script that creates the control panel that the user sees. The TychoTarget is documented in `$PTOLEMY/demo/whats_new/whats_new0.7/tychotarget.html`.