**TCS Description and Testing (June 2013)**

This document briefly describes the process involved in translating solar performance model code to C++ and outlines the procedure used to verify the code for the empirical trough model. Other technology models will follow similar methods. This document assumes an understanding of the SAM code and the SSC (system simulation core) interoperation layer.

### Methodology

The SAM team and ad hoc members have translated individual models of FORTRAN code that make up the empirical trough model in SAM. Each module has a ‘type’ number that identifies the purpose of the module. The empirical trough model uses three main types: Type 805 (the empirical trough module), Type 806 (the thermal energy storage module), and Type 807 (the energy conversion plant module). In addition, the empirical trough model uses the weather file reader module which simply opens resource files and retrieves resource information (like temperatures, wind speeds, solar radiation, etc.) for each hour of the simulation. All these modules work together in the code to simulation the plant operation over the year, so each module has to be fully translated before the model can work.

In addition, these modules need to operate within a framework than can call them iteratively for each time period of the simulation. Aron Dobos created the Transient Component Simulator (TCS) as this framework in 2012. The TCS framework controls the simulation by running individual models, managing the iterations for each time step, tracking and saving the input/output values for each module, and reporting overall progress.

Aron created the TCS project with a test environment that could run the components through a simulation. The test environment allowed developers to create lk scripts to setup runs and look through the outputs. This environment allowed developers to test the code that they had translated to make sure it ran properly (debugging); however, it was not an ideal environment to compare the results with the results from the original FORTRAN code (verification). SAM would be a much better environment for the verification because it allows a user to easily run the models over a range of input values and compare the results. Tom Ferguson was tasked with making the TCS model run from SAM.

Tom tested several methods for calling TCS framework from SAM, but ultimately settled on the simplest method he tried. Using this method saved development time, which seemed desirable since this was not meant to be a permanent solution for the chosen capability. Tom implemented all of the changes described below in a new branch of the SSC project (“efmsvn.nrel.gov/ssc/svn/branches/tcsapi”) in the SVN. So the changes below refer to changes from the original SSC Visual C++ (VC) project as implemented in the branch project.

To simplify and standardize the method by which SAM programmers can interact with the TCS, Tom created a wrapper class “tckernel” that makes the modules easier to interact with from the SAM programming environment. This class is derived from both the main TCS class “tcskernel” and the class that SAM uses to interact with modules in the System Simulation Core (SSC) “compute\_module”. This class can then simplify the code a developer needs to write to get inputs from the SAM model and send results back.

### Example technology

What follows is an example of creating a module that uses the TCS kernel to run a simulation from SAM using the empirical trough technology.

The programmer must first create a new module in the TCS SSC in the same way one is created in the SSC. First, add a new entry point in the sscapi.cpp file:

extern module\_entry\_info

/\* extern declarations of modules for linking \*/

cm\_entry\_6parsolve,

. . .

cm\_entry\_windpower,

cm\_entry\_geothermalui,

cm\_entry\_tcstrough;

/\* official module table \*/

static module\_entry\_info \*module\_table[] = {

&cm\_entry\_6parsolve,

. . .

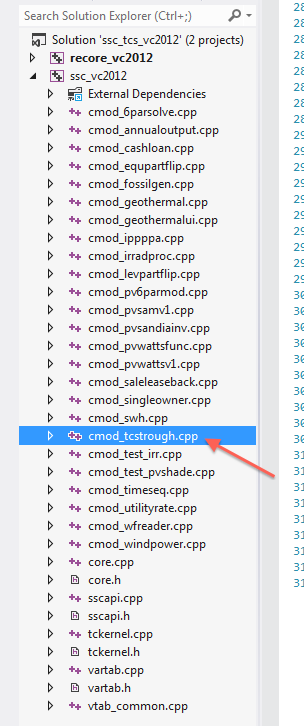
&cm\_entry\_windpower,

&cm\_entry\_geothermalui,

&cm\_entry\_tcstrough,

0 };

Next, define the module by creating a new class and its variable table, and defining its entry using the “DEFINE\_MODULE\_ENTRY” macro. This is usually done in a separate file named *cmod\_XXXXX.cpp* where the XXXXX represents the name of the module. In this example, I added a file called *cmod\_tcstrough.cpp*:



This file should be very similar to the module definition file used for the SSC technologies. The differences appear below:

1. When defining the variable table, it will be very helpful to make the variable names match those of the parameters defined in the type library used by the tcskernel. This way the programmer can avoid having to translate between the variable name used in the SSC to communicate with SAM and the variable name defined in the units of the type library.
2. The class for the module **has to be derived from the** tcKernel **wrapper class** (instead of the compute\_module class) in order to use the TCS capabilities:

//Definition of class used to interact with the TCS empirical trough model

class cm\_tcstrough : public tcKernel

{

public:

. . .

1. The constructor for the class should add a value to the tcskernel search path (using “add\_search\_path()”), in addition to calling the “add\_var\_info()” function. This tells the tcskernel object where to look for type libraries. For my tests, I created a copy of the typelib dll file in the sam deploy directory, the same directory as the ssc dll file. As such, I only added a single search directory – the current directory:

//Constructor for the cm\_tcstrough class

cm\_tcstrough()

{

add\_var\_info( \_cm\_vtab\_tcstrough );

add\_search\_path( "." );

}

1. The first order of business in the exec() function should be to load the type library into the tcskernel using the “load\_library()” function, or return an error if it cannot be loaded:

void exec( ) throw( general\_error )

{

if ( 0 >= load\_library("typelib") )

throw exec\_error( "tcstrough", util::format("could not load the tcs type library.") );

//Body of exec code follows...

. . .

1. Finally, the body of the exec function can now use both native tcskernel functions (e.g. “add\_unit”) and functions defined in the tcKernel class (e.g. “set\_unit\_value\_ssc\_array”) to interact with the TCS simulation class. Below is a partial listing of the *cmod\_tcstrough.cpp* file showing some of these functions:

//Add weather file reader unit

int weather = add\_unit("weatherreader", "TCS weather reader");

//Add Empirical Solar Field Model

int u1 = add\_unit( "sam\_trough\_model\_type805", "Test Trough" );

. . .

//Set weatherreader parameters

set\_unit\_value\_ssc\_string( weather, "file\_name" );

. . .

//Connect Solar Field Inputs

bool bConnected = connect( weather, "solazi", u1, "SolarAz", 0.1, -1 );

bConnected &= connect( weather, "beam", u1, "Insol\_Beam\_Normal", 0.1, -1 );

bConnected &= connect( weather, "tdry", u1, "AmbientTemperature", 0.1, -1 );

bConnected &= connect( weather, "wspd", u1, "WndSpd", 0.1, -1 );

. . .

// check if all connections worked

if ( !bConnected )

throw exec\_error( "tcstrough", util::format("there was a problem connecting units in the simulation.") );

// Run simulation

size\_t hours = 8760;

if (0 > simulate(3600, hours\*3600, 3600) )

throw exec\_error( "tcstrough", util::format("there was a problem simulating in tcstrough.") );

. . .

// get the outputs

set\_output\_array("enet", hours);

set\_output\_array("egr", hours);

Notice that the “simulate” function expects seconds as inputs. The lk scripts and TCS GUI program translate hours into seconds automatically. It must be done explicitly here – this allows for the possibility of sub hourly simulations.

At this point in the development, there are 7 helper functions defined in the tcKernel class specifically created to help programmers transfer data to/from the TCS kernel from/to the SSC interface. There are five for translating inputs from SAM into inputs for the units defined in the TCS, and two to help retrieve results from the TCS and send them back to SAM. The declarations of 4 of the functions follow:

void set\_unit\_value\_ssc\_string( int id, const char \*name );

void set\_unit\_value\_ssc\_double( int id, const char \*name );

void set\_unit\_value\_ssc\_array( int id, const char \*name );

void set\_unit\_value\_ssc\_matrix( int id, const char \*name );

These functions all follow the same format. Basically, they will look for an input in the TCS unit defined by “id” with the name “\*name” and fill it with the SSC\_INPUT variable with the same name. If the SSC variable type doesn’t match the type expected by the function, the SSC code will throw an error.

The other helper function for inputs has a slightly different format:

void set\_unit\_value\_ssc\_double( int id, const char \*name, double x );

This function was created to help with debugging – it makes it easy to set an exact value for a TCS unit input. It can be used like the “set\_value” function in the lk scripts used to run TCS.

The last two helper functions get data from the TCS results stored in the tcKernel class and translate them into SSC\_OUTPUT arrays to be sent back to SAM:

void set\_output\_array(const char \*output\_name, size\_t len);

void set\_output\_array(const char \*ssc\_output\_name, const char \*tcs\_output\_name, size\_t len);

The first function assumes the name of the TCS value and the SSC value are the same, it will look for a TCS\_NUMBER type parameter in any of the units matching “name” and set the SSC output array with the same name to that value for each step of the simulation. The second function serves the same purpose, but allows a programmer to use different names for the TCS parameter and the SSC array.

### Within SAM

The process for adding a technology that uses this method for running a TCS simulation from within SAM is the same as the process for adding any SSC interop model. This process is covered in the SSC documentation.

### Verification of the empirical trough model

The first steps for verifying the TCS empirical trough model from SAM were to make sure that it produced the same results as it did when called from within the TCS test environment. To accomplish this, Tom first ran the model with the inputs “hardwired” in the code to make sure it produced the same results. Tom then changed the code to get inputs from the SAM interface. The first few runs did not match and Tom had to figure out which of the SAM default inputs did not match the default inputs used in the TCS environment. Once this was done, again the results matched (the root mean squared deviation, RMSD, was under 0.001%).

With this done, Tom could be sure that SAM was running the TCS model correctly. Tom then started to compare the TCS results with the TRNSYS results. The results were close, but did not match exactly. Tom conferred with Ty Neises and determined that the cause was due to a known issue regarding the way that the FORTRAN code read the data in the weather files (the resource data). After accounting for this difference, the results matched closely: the RMSD as a percentage of the max output was about 0.05%, well under the goal of 1%.

For verification, Tom has started running parametric cases to determine if the TCS model deviates from the original FORTRAN code. At this point, having tested several different inputs across a range of values, the outputs match to within 0.01%.