Engineering Toolkit (ETK) SConsTools (SCT) Cantata Integration

Author: Chris Winstanley

# Overview

The ETK SCT provides a set of common SCons scripts and configuration files that all algo component projects can use to build their PC and target software. This document describes the integration of the ETK SCT with the Cantata toolchain, providing unit testing facilities for component projects.

This document contains the following sections:

1 Overview  
2 Installation  
3 Where To Find Things  
4 Usage  
5 Troubleshooting  
6 Configurable Options  
7 Customisation  
8 Hints and Tips

# Installation

## Prerequisites

You must have Cantata, Microsoft Visual Studio 2005, 2008 or 2010 and the Cantata MSVC plugin installed. For Cantata 6.2 follow the instructions here:

<https://workspace1.conti.de/content/00002191/Lists/CantataKnowledgeBase/DispForm.aspx?ID=41&Source=https%3A%2F%2Fworkspace1%2Econti%2Ede%2Fcontent%2F00002191%2FLists%2FCantataKnowledgeBase%2FAllItems%2Easpx>

## SCT Scons

The SCons Cantata integration is built into the ETK SConsTools shared sub-project. The ETK SConsTools sub-project is typically shared within your project in the 04\_Engineering/02\_Development\_Tools/scons\_tools directory.

If you are using the standard scons.bat file from the ETK SConsTools, it will automatically copy various generic SConscripts from 04\_Engineeing/02\_Development\_Tools/scons\_tools to the correct locations in your sandbox. This includes the many of the Cantata SConscripts.

However, there are some files that are project specific, that you will need to copy manually from the scons\_tools directory into the correct locations, and then modify for your project.

The installation steps are as follows:

1. Copy CantataWorkspacePreferences.epf and unit\_test.scfg from scons\_templates\05\_Testing\05\_Test\_Environment\algo\modtests\cantata\_tests\xxx to 05\_Testing\05\_Test\_Environment\algo\modtests\cantata\_tests\xxx.
2. Copy the SConscript.py (or merge if you have already modified this file) from scons\_templates\03\_Workspace\algo\xxx\_sim to 04\_Engineering\03\_Workspace\algo\xxx\_sim.
3. Copy the sconstruct\_config.scfg (or merge if you have already modified this file) from scons\_templates\03\_Workspace\algo\xxx to 04\_Engineering\03\_Workspace\algo\xxx
4. Execute scons.bat (with any target) to copy the generic scripts to the correct location in your sandbox.

If you need to tailor any of the generic files from the SCT\_SconsTools directory for your project you can remove them from the sconscript\_setup\_config.scfg file in the 04\_Engineering\03\_Workspace\algo\xxx directory so the scons.bat file does not overwrite them with the generic versions.

This installation only needs to be done once for the component project. All sandboxes created thereafter will contain the integration.

## EPF Preferences Manager

The EPF Preferences Manager is an Eclipse plugin. As Cantata is based on Eclipse we can make use of this plug in to automatically configure our Cantata workspace and projects with the appropriate settings, such as code coverage level, from within a SCons script.

The EPF Preferences Manager is held as a shared sub-project under the ETK:

/nfs/projekte1/REPOSITORY/Base\_Development/05\_Algorithm/ETK\_EngineeringToolKit/04\_Engineering/CANT\_Cantata/WorkspacePreferences

The sub-project should be shared under:

05\_Testing/05\_Test\_Environment/algo/modtests/cantata\_tests/xxx

To install the Preferences Manager follow the steps below. Note, this technique only works for Cantata 6.2 (Eclipse 3.7.1) onwards.

1. Start Cantata with any workspace;
2. Select Help->Install New Software….;
3. Press the Add button;
4. Press the Local button;
5. Select the directory:  
   05\_Testing/05\_Test\_Environment/algo/modtests/cantata\_tests/xxx/WorkspacePreferences
6. Press the OK button;
7. Untick the “Group items by category” box;
8. You should now see “Feature” in the Name box. Select it;
9. Keep pressing Next until you get to the license details and accept them;
10. Press Finish.

For Cantata 6.0 (Eclipse 3.5.2) simply copy the file com.ibm.rational.support.anttasks.epf.preferencesmanager\_1.0.1.jar into the plugins directory of the Cantata installation.

The installation of the Preferences Manager needs to be done once per Cantata installation.

# Where to Find Things

The directory structure of the Cantata integration is given below.

[SCT\_Sconstools] (normally shared into 04\_Engineering/02­\_Development\_Tools)  
[scons\_adas\_extensions]  
 eclipse\_cdt.py (Eclipse workspace and project generation facilities)  
 unittest.py (unit test build and execution facilities)

[04\_Engineering/03\_Workspace/algo]  
[<component\_name>]  
 SConstruct (entry point for all builds, including unit tests, shared from SCT\_Sconstools)  
 sconstruct\_sconfig.scfg (configuration file, the name of your component is defined here)  
[<component\_name>\_sim  
 SConscript.py (PC sim builds, shared from SCT\_Sconstools, ensures Cantata compiler not used)  
 [sim\_swc\_<component\_name>]  
 simenv\_config.scfg (contains PC compiler/linker flags, used by Cantata integration for builds)

[05\_Testing/05\_Test\_Environment/algo/modtests]  
 [cantata\_tests/<component\_name>]  
 [<unit\_under\_test\_A>] (test for a particular unit)  
 .project/.cproject (Cantata Eclipse project, generated, see section 4.2)  
 .imported (indicates project imported into Eclipse workspace, generated)   
 ipg.cop (settings for this particular unit test)  
 <unit\_under\_test\_A>.csi (Cantata instrumentation file, generated, see section 4.3.1/4.4)  
 test\_<unit\_under\_test\_A>.cpp (C or C++ testscript)  
 test\_<unit\_under\_test\_A>.csi (Cantata instrumentation file, generated, see section 4.3.1/4.4)  
 test\_<unit\_under\_test\_A>.cov (Cantata coverage results, generated, see section 4.4)  
 test\_<unit\_under\_test\_A>.ctg (Cantata , generated, see section 4.4)  
 test\_<unit\_under\_test\_A>.ctr (Cantata test results, generated, see section 4.4)  
 test\_<unit\_under\_test\_A>.vcproj MSVC project, generated, see section 4.2)  
 [<unit\_under\_test\_B>] (another unit test, see above)  
 [Framework] (see sections 6.1.3 and 8.1)  
 [workspace] (Cantata Eclipse workspace, see section 4.2)  
 [WorkspacePreferences] (Preferences Manager and component specific preferences  
 .epf file, see sections 2.3 and 6.2)  
 modtests.sln (MSVC solution, generated, see section 4.2)  
 SConscript.py (Entry point for unit test IDE generation, build and execution,  
 shared from SCT\_Sconstools)  
 SConscript\_unittests.py (Core functions for the Cantata integration, shared from  
 SCT\_Sconstools)  
 unit\_test.scfg (configurable options for the Cantata/SCons integration, see  
 section 6.1)

# Usage

Typical usage of the SCons Cantata integration follows a four step lifecycle:

1. Add a new module test to the appropriate configuration file
2. (Re)Generate the Cantata and MSVC IDE’s
3. Generate a test script
4. Modify test script, build and execute
5. View the results

## An Important Note about Licensing

The Cantata SCons integration gives you the option to work either in the Cantata Eclipse IDE or the MSVC IDE. The advantage of working in the Cantata Eclipse IDE is that you can do everything (except debugging) in one environment: generate test scripts, build, execute and browse test and coverage results using the built in result explorers. In MSVC you can only build and execute test scripts and browse the textual results.

The disadvantage of using the Cantata IDE is that it ties up one license for the duration that you are using the IDE and licenses are limited. However, the MSVC environment only ties up a license for the duration of the compilation and for a few minutes after compilation has completed.

In times of high usage, please use the MSVC IDE (or command line). You can check the current license situation using the Sentinel LM Admin tool in the Cantata installation.

## Adding a New Module Test

### General Case

Edit the file:

05\_Testing/05\_Test\_Environment/algo/modtests/cantata\_tests/xxx/unit\_test.scfg

At the end of the file you will find an array “utprograms”. Add the filename of the unit to be tested to this array, without any file extension. For example:

utprograms = ['LDCamera',  
 'LDLTMHState',  
 'LDPlausibilityMeasures',  
 'LDPMCamera',  
 'LDPMRoadStructures',  
 'LDPMVehicleDynamics',  
 'LDTravelJournal',  
 'your\_new\_module\_here']

By default the source code to be tested will be taken from:

04\_Engineering/01\_Source\_Code/algo/xxx

### Sub-Folders

Sub-folders are supported, for example:

utprograms = ['Camera/LDCamera',  
 'State/LDLTMHState',  
 'Plausi/LDPlausibilityMeasures']

will take the source code from the sub-directories:

04\_Engineering/01\_Source\_Code/algo/xxx/Camera  
 04\_Engineering/01\_Source\_Code/algo/xxx/State  
 04\_Engineering/01\_Source\_Code/algo/xxx/Plausi

respectively.

### Header Files Without Bodies

Cantata only generates test scripts for source file bodies, i.e. “.c” and “.cpp” files. To test a header file that has no associated body, you have to create a dummy body, for which you can generate a test script.

Dummy bodies can be created in a sub-directory under the test directory:

05\_Testing/05\_Test\_Environment/algo/modtests/cantata\_tests/xxx

The exact sub-directory is specified in the unit\_test.scfg file, for example:

dummy\_src\_dir = "DummySrc/"

specifies the directory:

05\_Testing/05\_Test\_Environment/algo/modtests/cantata\_tests/xxx/DummySrc

You can then use the “$” prefix to specify that the dummy source code body should be taken from the dummy\_src\_dir:

utprograms = ['$LDCamera',  
 '$LDLTMHState',  
 '$LDPlausibilityMeasures']

## Generating the Cantata and MSVC IDE’s

Call the scons batch script with the appropriate target:

scons.bat <your\_component\_name>\_ide\_unittests

For example, for the Preview project (pv):

scons.bat pv\_ide\_unittests

The component name and constituent algorithm names should already be setup in:

03\_Workspace/algo/<component\_name>/sconstruct\_config.scfg

This will generate:

* A Cantata Eclipse workspace containing:
  + A Cantata project per module under test
    - A configuration per build type (debug, release) and algorithm
* A MSVC solution containing:
  + A MSVC project per module under test
    - A configuration per build type (debug, release) and algorithm

During the generation of the Cantata IDE, workspace preferences are imported from:

05\_Testing/05\_Test\_Environment/algo/modtests/cantata\_tests/xxx/  
 CantataWorkspacePreferences.epf

The workspace preferences include settings for the level of code coverage gathered. See sections 2.3 and 6.2 for more information.

To open the Cantata unit test environment:

1. Start Cantata
2. At the workspace prompt select:  
   05\_Testing/05\_Test\_Environment/algo/modtests/cantata\_tests/xxx/workspace

To open the MSVC environment, double click on:

05\_Testing/05\_Test\_Environment/algo/modtests/cantata\_tests/xxx/modtests.sln

To clean the IDE’s simply:

scons.bat –c pv\_ide\_unittests

## Generating a Test Script

Generating a test script is a two step process.

### Step 1: Generate the CSI File and ipg.cop File

To generate a test script for a unit test, you must first generate a .csi file and an ipg.cop file. The .csi file is an instrumented version of the file under test. The ipg.cop file contains configurable options for a particular unit test. The generation of these files is done automatically, the first time you build the project for a particular unit test.

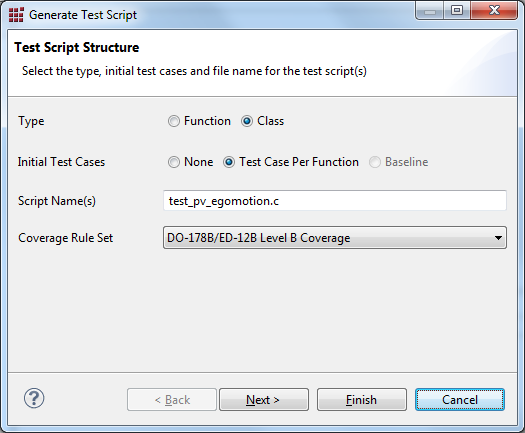
The ipg.cop will be generated with a default set of options to which you can add later. The ipg.cop file is only generated once. Once generated, it will never be generated again unless you delete it. This is so you can add options to the file without fear of it being overwritten.

To generate these files, open either IDE and simply build the project.

### Step 2: Generate the Test Script

Generation of the test script can only be done within the Cantata environment. Open the Cantata IDE. You should now be able to see the .csi and ipg.cop file within the project.

Right click on the project, Cantata->Generate Test Script(s)….



Type:  
For C++ projects you have the option of creating a test class. For C you can only create test functions.  
  
Initial Test Cases:  
For C and C++ test scripts you can choose to generate an empty test script (None), or a test case function per function in the file under test. Additionally, for C projects, you can generate a Baseline test. For further information, see the Cantata user manual.

Script Name(s):  
The name of the script to be generated. The Cantata SCons integration assumes the name will be “test\_<file\_under\_test>”, i.e. the default given. Do not change the name.

Coverage Rule Set:  
By default DO178B Level B Coverage is selected. This is suitable for 26262 ASIL B. If you wish to override this setting for a single unit test, modify it here. However, if you wish to override this setting for all unit tests in the project, it should be done by modifying the workspace preferences – see section 6.2.

To select exactly which stubs and wraps are created, press the Next button, make your selections and then press the Finish button. To accept the default selection of stubs and wraps, simply press the Finish button.

The test script is now generated.

## Build and Execution

Compile and link options for the unit tests are drawn from the compile and link options specified for the PC simulation build of the component. These flags are normally specified in:

04\_Engineering/03\_Workspace/algo/<component\_name>\_sim/  
 sim\_swc\_<component\_name>/simenv\_config.scfg

Every time you modify the test script or file under test and build, the instrumented version of the file under test (.csi) will be rebuilt.

At the end of build and execution, test results files (.ctr and .ctg) and a coverage results file (.cov) will be produced. The .ctr file is a human readable text file but the .ctg and .cov files are not human readable and must be viewed by via the Cantata IDE (see section 4.6).

### Cantata IDE

To build in the Cantata IDE, start Cantata and open the workspace as described in section 4.2. Right click on the project and select Build Project.

The unit test will be built and automatically executed. The output of the build and a summary of the test results can be seen in the Console tab. Errors and warnings can be seen in the Problems tab.

Selection of debug or release builds can be performed by a right click on the project, select Build Configurations -> Set Active. For multi-algorithm components, selection of the algorithm to build can also be performed by this mechanism.

To clean the project, right click on the project and select Clean Project.

Note, at present, debugging can only be performed in the MSVC IDE.

### MSVC IDE

Open the MSVC solution as described in section 4.2. Select the project and build and clean with your preferred technique (right click or icon or menu item).

The unit test will be built and automatically executed. The output of the build and a summary of the test results can be seen in the Output tab. Errors and warnings can be seen in the Error List tab.

Selection of debug or release builds can be performed by choosing from the Configuration drop down list. For multi-algorithm components, selection of the algorithm to build can also be performed by this mechanism.

### Command Line

To build and execute all unit tests:

scons.bat <component\_name>\_unittests\_<build\_type>

For example for the Preview project in debug mode:

scons.bat pv\_unittests\_debug

A summary of which unit tests have passed and which unit tests have files will be output as the end of the run. If there is a compilation error, the run will terminate immediately.

To build and execute a single unit test:

scons.bat <filename\_without\_file\_extension>\_ut\_<build\_type>

For example, to build and execute the test for the file pv\_egomotion.c in debug mode:

scons.bat pv\_egomotion\_ut\_debug

To clean all unit tests:

scons.bat –c <component\_name>\_unittests\_<build\_type>

To clean an individual unit test:

scons.bat –c <filename\_without\_file\_extension>\_ut\_<build\_type>

## Viewing the Results

### Textual Output

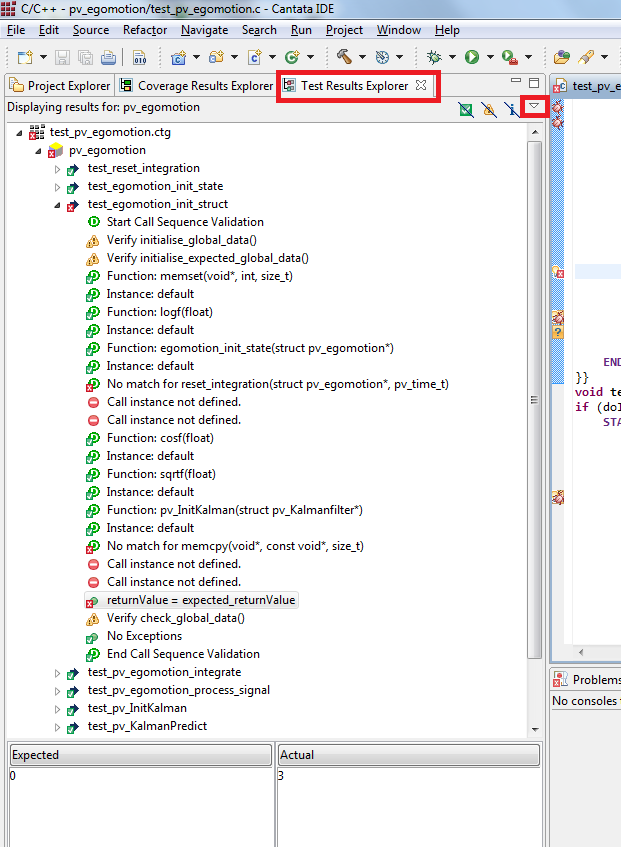
A textual representation of the results is stored in the <unit\_under\_test>.ctr file. It contains results for each individual test case and also coverage results, including details of non-covered statements and conditions.

### IDE Navigation

The test results and coverage results can also be viewed via the Cantata IDE.

To view the test results:

1. Start Cantata
2. Select the project, right click, select Cantata->Load Result Files.
3. Click on the Test Results Explorer tab
4. Select the project to view results for by clicking on the small white upside down triangle in the upper right hand corner of the Results Explorer
5. Expand the <unit\_under\_test>.ctg item in the explorer to view passed and failed items.
6. Double click on failed items to take you to the point in the test script where the test failed. Additional information on the failure is also given in a small pane at the bottom of the explorer window. The content of this pane is context senstitive.

Figure Test Results Explorer

To view the coverage results:

1. Start Cantata
2. Select the project, right click, select Cantata->Load Result Files.
3. Click on the Coverage Results Explorer tab
4. the project to view results for by clicking on the small white upside down triangle in the upper right hand corner of the Results Explorer
5. Expand the directory structure until you can see the unit under test.
6. Expand the unit under test to see the coverage of functions:  
   Green = fully covered  
   Yellow = partially covered  
   Red = not covered
7. Double click on a function to show precise details of the coverage in the Coverage Viewer tab.

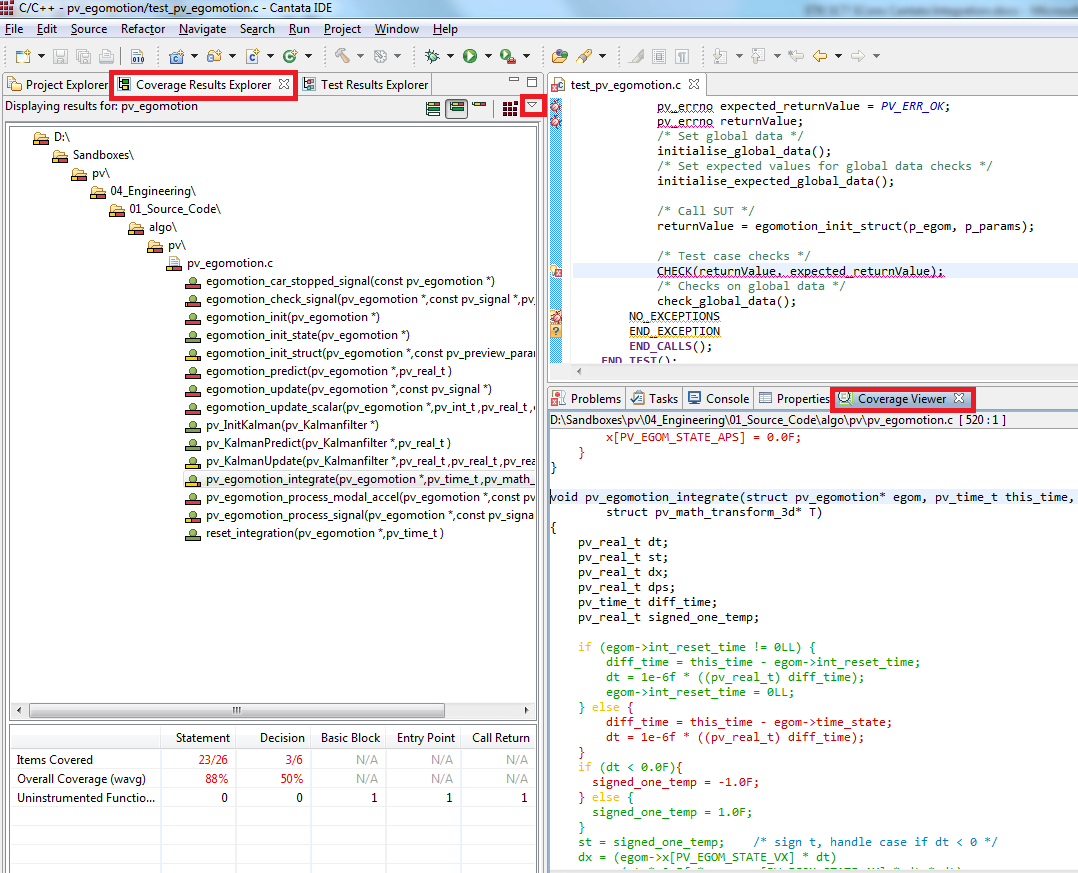


Figure Coverage Results Explorer

# Troubleshooting

## My Normal PC Simulation Builds Don’t Work Anymore

Problem: When you try to build your normal PC simulation environment it fails. It looks like it is trying to instrument it with Cantata.

Reason: When you build and execute a unit test, the ipg.cop file is copied into the root scons directory of your sandbox. This effectively enables Cantata for a build. If the unit test build fails in some way, the normal clean up of this file may not occur.

Solution: Delete the ipg.cop file from your 04\_Engineering/03\_Workspace/algo/<component\_name> directory.

## I’m Testing C Code but Cantata Is Generating C++ Stuff in My Test Script

Problem: When you generate a test script for C code, you can see the C++ options being offered in the generation dialogue box and the resulting test script contains things like START\_EXCEPTION, END\_EXCEPTION, giving warnings in the Cantata IDE.

Reason: Your project contains the /TP (compile as C++) flag in its compiler flags. Cantata parses the compiler command line and decides to generate C++ stuff in the test script.

Solution: Note, this does not stop your test script from compiling and executing, but it does result in a lot of annoying warnings in the Cantata IDE. These warnings are not compiler warnings, they are Cantata warnings. Edit the unit\_test.scfg file and enable the “disable\_tp\_flag” option. See section 6.1.4 below.

# Configurable Options

## The unit\_test.scfg File

The standard SCT/Cantata integration has a number of configurable options, accessed via the unit\_test.scfg file. This allows you to configure, for example, whether it is a C++ or C project, the locations of files, additional files to be included in every unit test, etc.

The unit\_test.scfg file can be found here:

05\_Testing/05\_Test\_Environment/algo/modtests/cantata\_tests/xxx

Do not delete any variables in this file, as they are required for the unit test SConscripts to execute correctly.

The following sections describe the configurable options.

### Common Paths

The locations of certain files and also the exact command used to execute scons can be configured here.

If you have a standard SCT installation that uses the scons.bat file as the entry point for SCons builds then you can leave the scons\_cmd variable as an empty string. In this case the unit test scripts will auto-discover the commands needed to execute SCons builds.

If you start SCons in some non-standard manner, then put the exact command line used to execute SCons in the scons\_cmd variable, including absolute paths. Do not, however, specify command line arguments.

### Targets

The unit\_test\_target variable specifies the target used to build and execute all unit tests. Modify as you see fit, but if you do don’t forget to update the help information in the scons.bat file.

### Source File Lists

#### Framework Headers

Here you can specify additional header files to be included every unit test project. This can be particularly useful when you have some generic stubs that you need for every test case.

For example, if you have a C++ project, where objects are commonly passed as parameters, you often need to provide stubs for many class constructors to allow the unit test to link, even if these objects are not directly used in the unit under test. This feature allows you to build a framework of support files, speeding up the development of unit tests.

Any files specified here, just be placed in the following directory:

05\_Testing/05\_Test\_Environment/algo/modtests/cantata\_tests/xxx/Framework

#### Test Script Source File Inclusion

When you are testing C code that contains static functions and global variables, by default you cannot access these items from the test script. A trick to avoid this problem is to include the .c file in the test script, instead of the .h file.

However, if you use this approach, you do not want to compile the source file under test separately and link it because then you will have multiple symbol definitions for everything in the source file under test.

Therefore, the configuration file provides a switch to allow you to exclude the file under test:

# Specifies if the source file under  
# test is included directly in the  
# test script.  
#  
# eg: #include "pv\_egomotion.c"  
#  
# If so, then the source file under  
# test is not compiled separately.  
#  
# This is particular of use to   
# C projects, where to gain access  
# to static functions and variables,  
# the .c file is included in the test  
# script, instead of the .h file.  
# = 0, source file not included in test script  
# = 1, source file included in test script  
source\_included\_in\_test\_script = 0

For C++, this trick is not needed as you can simply add a “friend” statement to the ipg.cop file.

Since, unit\_tests.scfg 1.6 and Sconscript\_unittests.püy 1.28 there’s a setting to configure this option for each Cantata test script separately

# 'source\_included\_in\_test\_script' dictionary to store flag for each utprogram  
# or use 'source\_included\_in\_test\_script' as default  
# Note: This will influence the ‘static variable access’ and ‘static function access’ features

utprograms\_source\_included\_in\_test\_script = {}

For example

source\_included\_in\_test\_script = 0  
utprograms\_source\_included\_in\_test\_script = {  
 'frame/pfc\_main': 1  
}

enables the ‘source\_include\_in\_test\_script’ option only for ‘frame/pfc\_main’ test script and keeps the default value for all other test scripts.

### Compiler Configuration

Specify whether this is a C or C++ project by setting the file extension for source files.  
source\_ext = '.cpp'  
#source\_ext = '.c'  
Since check point AL\_ETK\_SCT\_01.02.00\_RELEASE (1.37), source\_ext is replaced by  
**source\_ext\_list = [".c", ".cpp"]**  
which contains a list of supported file extsionsions for C and C++ files.

Also the object file extension can be specified here:  
object\_ext = '.obj'

Specify whether to disable the /TP compiler flag for unit test builds. The /TP compiler option compiles C code as if it were C++ code. When generating tests for C code using Cantata, Cantata becomes confused and offers C++ options when generating C test scripts (see section 5.2 above).

When testing C code it is recommended that you enable the “disable\_tp\_flag” option by setting it to 1.

Specify additional defines for a unit test. An array of values can be provided, indexed by items in the algo\_name\_list. The algo\_name\_list (a list of algorithms contained within the component) is specified in:

04\_Engineering\03\_Workspace\algo\xxx\sconstruct\_config

For example:

ut\_additional\_defines = [["CB\_MODULE"], ["SCB\_MODULE"]]

Here, an additional define is specified for each algorithm in the camera blockage component (mono and stereo algorithms). For components that contain only one algorithm, you can specify for example:

ut\_additional\_defines = [["CHECK\_INVARIANTS"]]

### HTML Reports

Automatic generation of HTML reports can be turned on and off with the

generate\_html\_reports = 1

flag. 1 for turning on, 0=off.

There’s an additional option (0 or 1) to turn on/off the usage of the workspace option in the HTML report generation command. It should be turned on if large test report files (.ctr) > 100MB are used.

generate\_html\_use\_data\_option

HTML reports are stored under:

05\_Testing\02\_Reports\algo\modtests\cantata\_tests\xxx

### List of Unit Tests

The utprograms variable contains a list of all unit currently tested. Each entry in the list should be the filename of the unit under test, without a file extension.

## Export of Cantata Settings

Cantata is built upon Eclipse. The settings for an Eclipse workspace can be exported to a preferences file (.epf) and imported into another workspace. We make use of this facility to create a standard set of Cantata settings (for example, coverage level) that can be imported into the Cantata workspace generated by the Cantata SCons integration.

If you wish to tailor the default settings provided you can do so by making the appropriate settings within the Cantata IDE and then regenerating the .epf file:

1. File->Export…
2. Expand the General folder
3. Select Preferences in the resulting list
4. Press the Next button
5. Tick the checkbox next to Cantata
6. Browse to the location you want to store the preferences and specify a filename
7. Press the Finish button

Note that the path to the preferences file used by the SConscript\_unittests.py is specified in the SetupIde function.

# Customisation

If the configuration options described in section 6 above offer insufficient flexibility, for example, if the standard SCT sconscripts have been significantly tailored for your project, then customisation of the unit test sconscripts may also be necessary.

This section describes the contents of the various SConscripts that contain unit test functionality. This will provide you with the knowledge you need to adapt the scripts to non-standard SCons installations or to extend the unit test functionality to more closely fit the needs of your component.

## 03\_Workspace/algo/<component\_name>/SConstruct.py

This is the entry point to the entire component build chain, including unit test functionality. The unit test SConscript is called from here, passing the component name:

# build and execute unit tests  
SConscript(testing\_dir + "05\_Test\_Environment/algo/" + unit\_test\_dir + "/cantata\_tests/"+component\_name+"/SConscript.py",   
 exports = {"component\_name" : component\_name})

## 03\_Workspace/algo/<component\_name>\_sim/SConscript.py

The Cantata tool can interfere with the operation of the PDO tool. When you install Cantata, it copies the MS cl.exe and link.exe to ms\_cl.exe and ms\_link.exe respectively. It then installs its own cl.exe and link.exe which enables Cantata to instrument the code under test. The Cantata cl.exe and link.exe call the original ms\_cl.exe and ms\_link.exe as the final step in the compile/link process.

The Cantata cl.exe always outputs a banner – this behavior cannot be controlled. This can cause problems if you are explicitly producing pre-processed output, for example when the PDO tool is used.

To work around this, in the original host SCons build environment, if Cantata is installed, we need to explicitly set the compiler used to be ms\_cl.exe. In the code below, pc\_env is the PC host environment:

# If cantata is installed, cl.exe and link.exe will have been  
# renamed to ms\_cl.exe and ms\_link.exe.  
# Cantata outputs a banner to stdout that cannot be suppressed -  
# this causes problems when preprocessing files in non-unit test  
# builds.Therefore, if Cantata is installed, we set the root PC  
# environment to use the original ms exe's, and override for the  
# unit test environment.

catd60 = Dir( cantata\_tool\_dir\_6\_0 ).abspath  
catd62 = Dir( cantata\_tool\_dir\_6\_2 ).abspath

if FindFile("cantpp.exe", catd62):  
 pc\_env['CC'] = 'ms\_cl'  
else:   
 if FindFile("cantpp.exe", catd60):  
 pc\_env['CC'] = 'ms\_cl'

## 05\_Testing/05\_Test\_Environment/algo/modtests/SConscript.py

To operate, Cantata needs specific environment variables, paths and compiler/linker settings. We separate these settings from the rest of the build environment by creating a unit test SCons environment within this SConscript. It extracts the debug compile and link options specified for the PC simulation environment in:

03\_Workspace/algo/<component\_name>\_sim/sim\_swc\_<component\_name>/  
 simenv\_config.scfg

and inserts them into the unit test environment.

By default, SCons specifies the /Z7 flag to ensure parallel builds work correctly. However, this can cause conflict if the component specifies the /Zi flag. As the unit test SConscripts ensure the unit tests are serialised anyway (see section 7.4.9 for more information),in this case the /Z7 flag is suppressed and the /Zi flag is used.

Finally it creates a variant directory for the unit test build artifacts and calls the core unit test SConscript, SConscript\_unittests.py.

## 05\_Testing/05\_Test\_Environment/algo/modtests/SConscript\_unittests.py

This SConscript contains the core SCons/Cantata integration. It contains functions to generate the IDE’s, build and execute the unit tests.

### Source File Lists, IDE Project Lists, Test Result Lists, Build Control

Specifies empty lists to be used in building the unit tests, the IDE’s and test results. Also declares some unit test build control variables.

### Function \_GatherTestResults

Builds a list of successful and failed tests for a summary report.

### Function \_PrintTestResults

Prints a summary report of the passed and failed tests.

### Build Actions GatherTestResults/PrintTestResults

Turns the \_GatherTestResults and \_PrintTestResults functions in to SCons build actions for execution during build and execution of the unit tests.

### Function CleanedSideEffect

Declares a side effect for a particular build target and ensures it is removed on a build clean. For example, files with a “.ti” are side effects of instrumenting the source code with Cantata.

### Function SetupUnitTestSources

Builds the list of source files to be used in each unit test and for creation of the IDE’s.

### Function ProcessUnitTestOutput

Copies the result files generated by unit test runs from the variant build directory to the appropriate unit test project directory. This is so the results can then be viewed correctly within the Cantata IDE.

### Function SetupIde

Generates the IDE solutions, workspaces and projects for Visual Studio and Eclipse (Cantata).

### Function BuildAndRunUnitTest

Builds the unit test from its sources and executes it. Ensures that the build and execution of unit tests is serialized as the ipg.cop file for each unit test must be copied to the root SCons directory before execution, to enable Cantata for the build. Thus unit tests cannot be run in parallel.

### Function \_GenerateIpgCop

Generates an ipg.cop file for a unit test on the very first compile. Never regenerated. Coverage options are specified here which you may want to tailor. Also, specifies an option that forces the output of the instrumented version of the file under test to the unit test directory (otherwise it is created in the source file directory by default).

### Build Action GenerateIpgCop

Turns the \_GenerateIpgCop function into a SCons build action for execution during build and execution of the unit tests.

### Function GenerateCopAndCsi

Generates the ipg.cop and .csi files on first compile of the unit under tests. This then allows generation of a test script within the Cantata environment.

### Function ReadFileExtsions

Checks the file extension of the source file under test and stores to extension to use the same one to generate the Cantata test script.

### Main

This is the entry point for the SConscript file.

#### Scons Command and Arguments

To be able to generate the MS Visual Studio solution and projects we need to know the SCons command and its arguments so Visual Studio can call SCons to build and execute the unit tests:

ide\_scons\_cmd = (r"cd $SCONS\_DIR && scons.bat -Q -j " +   
 str(thread\_number) + r" $SCONS\_ARGUMENTS")

#### Specification of Tool Directory

Setups the path to the Cantata installation to allow execution of the Cantata tools. The path differs depending upon whether Canatata 6.0 or 6.2 has been installed:

catd = None  
if FindFile("cantpp.exe", catd62):  
 catd = catd62  
else:   
 if FindFile("cantpp.exe", catd60):  
 catd = catd60  
  
ut\_env.AppendENVPath('PATH', catd)  
ut\_env.AppendENVPath('PATH', catd+"\\bin")

#### License Server

Cantata obtains the license server from the LSFORCEHOST environment variable. We make this available within the SCons unit test environment:

try:  
 pc\_env\_ut.AppendENVPath('LSFORCEHOST', os.environ['LSFORCEHOST'])  
except:  
 print "WARNING! LSFORCEHOST environment variable not set."  
 print "Cantata based unit tests will not build."

#### Building a Console App

Cantata test scripts run as console applications. The PC host environment will probably be a windows application. Therefore we need to override the appropriate settings to create a console application and remove the DLL flag:

arflags = str(pc\_env\_ut['ARFLAGS'])  
arflags = arflags.replace('/SUBSYSTEM:WINDOWS', '/SUBSYSTEM:CONSOLE')  
ut\_env.Replace(ARFLAGS=arflags)  
ut\_env.Append(ARFLAGS=' /SUBSYSTEM:CONSOLE')  
linkflags = str(ut\_env['LINKFLAGS'])  
linkflags = linkflags.replace('/SUBSYSTEM:WINDOWS', '')  
linkflags = linkflags.replace('/DLL', '')  
ut\_env.Replace(LINKFLAGS=linkflags)  
ut\_env.Append(LINKFLAGS=' /SUBSYSTEM:CONSOLE')

#### Build IDE’s, Unit Tests and Output Summary Results

Finally the IDE’s are built, the unit tests are built and executed and a summary of the test results is produced by calling the functions described above. The build is only performed if a Cantata installation could be found, otherwise a warning message is displayed.

# Hints and Tips

## Building a Unit Test Framework

### Common Stub Files

During development of LD unit tests, I found that I was generating stubs for the same class constructors , virtual functions and static constants over and over again simply because they were needed to link the test executable but were not actually used directly by the unit under test. Rather than repeat these definitions in every test script, I created a set of common framework include files that could be included from each test script, providing these stubs.

You may find yourself in a similar position, in which case creating a similar framework for your project makes sense.

Within the LD we have:

* ConstructorStubs.h – stubs of all constructors within the project
* StaticConstStubs.h – stubs of static constants within the project
* VirtualStubs.h – concrete stubs of virtual functions within the project

You can find the LD framework files here:

Base\_Development/05\_Algorithm/LD\_LaneDetection/04\_Engineering/00\_CodeGen/design/  
unittests/Framework

The stubs themselves have a simple default implementation sufficient for link purposes. However, when you actually want to test one of these constructors or virtual functions directly then you need to exclude these generic stubs and provide an intelligent stub within your test script.

This is done simply by using conditional compilation. Within the framework stubs file you define something like:

**#ifndef TEST\_LDYAWAXIS  
C\_LDYawAxis::C\_LDYawAxis(S\_LDYawAxisState &rs\_State) :  
 m\_rs\_State(rs\_State),  
 C\_LDPitchYawAxis(rs\_State.m\_s\_Inherited)  
{  
 if (C\_TestScriptControl::pc\_instance()->b\_CallSequenceValidationEnabled())  
 {**  
 REGISTER\_CALL("C\_LDYawAxis::C\_LDYawAxis(S\_LDYawAxisState &)");  
  
 IF\_INSTANCE("default") {   
 return;  
 }  
  
 LOG\_SCRIPT\_ERROR("Call instance not defined.");  
 }  
 return;  
}  
#endif

Then at the top of your test script, if you want to test this constructor directly, you simply add the line:

#define TEST\_LDYAWAXIS

Then you provide an explicit stub within the test case.

### Fine Grain Test Control

Cantata provides a mechanism called Call Sequence Validation (CSV). You provide a list of calls to Cantata of functions external to the unit under test that will be called by the unit under test and thus are stubbed. For more information see the Cantata user manual.

Cantata allows you to turn CSV on and off at the start and end of a test case. However, this can sometimes be a little restrictive. For example, if you have to construct an object to pass to the function under test, the construction can result in calls that you are really not interested in validating.

For these cases, you can include the TestScriptControl.h file in your test script. This allows you to extend your stubs so they can choose to take part in CSV or not.

You can find the TestScriptControl.h file here:

Base\_Development/05\_Algorithm/LD\_LaneDetection/04\_Engineering/00\_CodeGen/design/  
unittests/Framework

So for example, in your test script you can do something like:

C\_TestScriptControl::pc\_instance()->v\_SuppressCallSequenceValidation();  
C\_LaneDetectionAlgo\* pc\_LDAlgo = new C\_LaneDetectionAlgo(m\_s\_MasterState); C\_TestScriptControl::pc\_instance()->v\_EnableCallSequenceValidation();

Within the constructor stub you have something like:

if (C\_TestScriptControl::pc\_instance()->b\_CallSequenceValidationEnabled())  
 {  
 REGISTER\_CALL("C\_LaneDetectionAlgo::C\_LaneDetectionAlgo(S\_LaneDetectionAlgoState &)");  
  
 IF\_INSTANCE("default") {   
 return;  
 }  
  
 LOG\_SCRIPT\_ERROR("Call instance not defined.");  
 }

Then any calls made during the C\_LaneDetectionAlgo constructor are not validated.

### Including the Framework in your Test Script

Include the framework at the top of your test script like so:

#define TEST\_SCRIPT\_GENERATOR 2  
#define TEST\_LDPMROADSTRUCTURES

/\* Include files from software under test \*/  
#include "LaneDetectionAlgo.h"  
  
#include <cantpp.h> /\* Cantata++ Directives \*/  
  
#include "ConstructorStubs.h"  
#include "VirtualStubs.h"  
#include "StaticConstStubs.h"

Note the ordering is important. The framework files must come after the inclusion of the software under test header file as the framework files provide stub bodies to the declarations in the header file. They must also come after the inclusion of the cantpp.h file as they use Cantata facilities.

Tip: including the top level header file of your algo makes everything that you could possible need available within the test script.