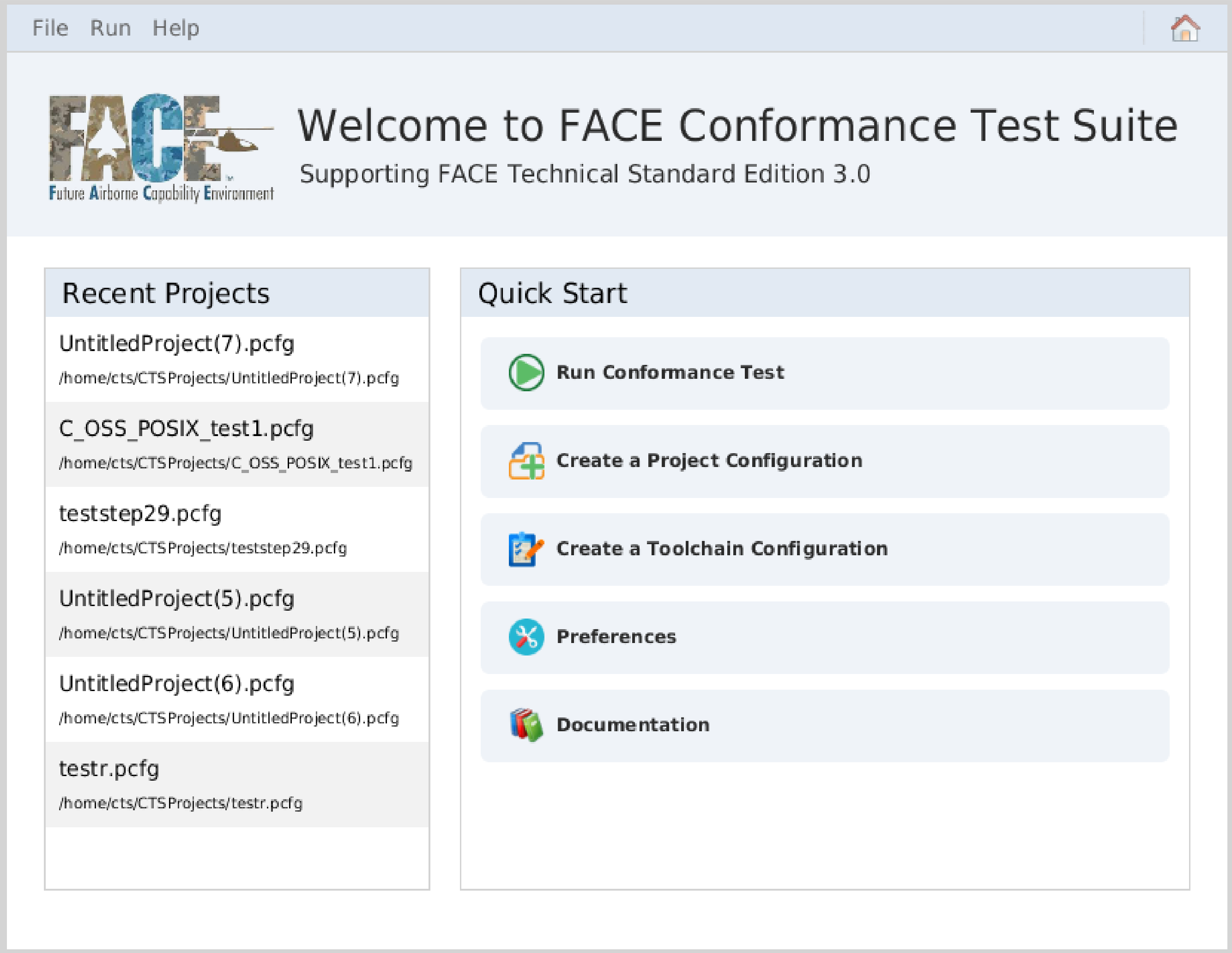
Conformance Test Suite Manual
for Testing Interface and Application
Code Against the FACE^{TM} Standard 3.0

Version 3.0.0

GTRI Document No. CTSD75666181-001



Welcome to FACE Conformance Test Suite

Supporting FACE Technical Standard Edition 3.0

Recent Projects

- UntitledProject(7).pcfg
  /home/cts/CTSProjects/UntitledProject(7).pcfg
- C_OSS_POSIX_test1.pcfg
  /home/cts/CTSProjects/C_OSS_POSIX_test1.pcfg
- teststep29.pcfg
  /home/cts/CTSProjects/teststep29.pcfg
- UntitledProject(5).pcfg
  /home/cts/CTSProjects/UntitledProject(5).pcfg
- UntitledProject(6).pcfg
  /home/cts/CTSProjects/UntitledProject(6).pcfg
- testr.pcfg
  /home/cts/CTSProjects/testr.pcfg

Quick Start
Run Conformance Test
Create a Project Configuration
Create a Toolchain Configuration
Preferences
Documentation

Georgia Tech Research Institute acknowledges The Open Group for permission to include text/figures derived from its copyrighted Future Airborne Capability Environment (FACE™) Reference Architecture. FACE is a trademark of The Open Group in the United States and other countries.

Acronyms Used

CR	Change Request
CTS	Conformance Test Suite
CVM	Conformance Verification Matrix
DSDM	Domain-Specific Data Model
FACE	Future Airborne Capability Environment
GTARC	Georgia Tech Applied Research Institute
GTRI	Georgia Tech Research Institute
GSL	Gold Standard Library
IOSS	Input/Output (I/O) Services Segment
LCM	Life Cycle Management
OCL	Object Constraint Language
OS	Operating System
OSS	Operating System Segment
PCS	Portable Components Segment
PSSS	Platform Specific Services Segment
SDM	Shared Data Model
TOG	The Open Group
TSS	Transport Services Segment
UoC	Unit of Conformance
UoP	Unit of Portability
USM	UoP Supplied Model
VA	Verification Authority

Introduction

The Conformance Test Suite tests the following Units of Conformance (UoCs) / Units of Portability (UoPs) for meeting a subset of the requirements in the FACE Technical Standard, Edition 3.0. The subset of requirements the test suite is responsible for testing is defined in the Conformance Verification Matrix (CVM) provided by the FACE Consortium.

1. Portable Components Segment (PCS) UoCs
2. Platform Specific Services Segment (PSSS) UoCs
3. Transport Services Segment (TSS) UoCs
4. I/O Services Segment (IOSS) UoCs
5. Operating System Segment (OSS) UoCs

Testing procedures for each segment are listed in the chapters below.

System Requirements - Linux (CENTOS/RHEL)

Before installation check the system requirements below to ensure the test suite will run on your designated machine. The software has been tested on CENTOS/RHEL 7. It is highly recommended to use this version of Linux.

Overview

Common requirements:

- Python version 2.7.x (not 3.x) with zlib support and setuptools ('setuptools' is the name of a Python tool. Ensure it is packaged with your Python installation.)
- Google Protocol Buffers version 2.6.0
- Java 1.7 runtime (be sure that Java is in your PATH)
- a PDF Viewer
- Recommended 4GB+ RAM

Language-specific requirements:

<u>C/C++:</u>

- GCC/G++ version 4.8+

<u>Ada:</u>

- GNAT for GCC version 4.8+

<u>Java:</u>

- Java Development Kit (JDK) 8
- Linux alternatives utility
- Ant 1.9.x
- Qt 5.2.1

Detailed Instructions for Installing Prerequisites

GCC/G++ 4.8.5

Execute the following command to install gcc/g++

```
$ sudo yum install gcc gcc-c++ gcc-gnat
```

This is necessary for C/C++ projects and for building and installing the Protocol Buffers library as described below.

Python 2.7

If Python 2.7 is not already installed, then install it for your OS following the instructions from https://www.python.org/downloads/

Note: If using CENTOS 7 or RHEL 7, the default version of Python is already 2.7.

Protocol Buffers 2.6

Download protobuf-2.6.0.tar.gz from location https://github.com/google/protobuf/releases/tag/v2.6.0

Unzip it with the following command:

```
$ tar -xvf protobuf-2.6.0.tar.gz
```

Navigate to the protobuf-2.6.0 folder and install Protocol Buffers with the following commands:

```
$ ./configure

$ make

$ sudo make install
```

Add the shared libraries to the search path with the following commands:

```
$ sudo su

# echo "/usr/local/lib" > /etc/ld.so.conf.d/local.conf

# exit

$ sudo ldconfig
```

Java 7 JDK

Download "jdk-7u80-linux-x64.rpm” from location http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html

Navigate to the directory where you downloaded the rpm and execute the following commands:

```
$ sudo yum install jdk-7u80-linux-x64.rpm

$ sudo /usr/sbin/alternatives --install /usr/bin/java java /usr/java/jdk1.7.0_80/bin/java 2000

$ sudo /usr/sbin/alternatives --install /usr/bin/javac javac /usr/java/jdk1.7.0_80/bin/javac 2000
```

Java 8 JDK

Download "jdk-8u161-linux-x64.rpm" from location http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Navigate to the directory where you downloaded the rpm and execute the following commands:

```
$ sudo yum install jdk-8u161-linux-x64.rpm

$ sudo /usr/sbin/alternatives --install /usr/bin/java java /usr/java/jdk1.8.0_161/bin/java 2000

$ sudo /usr/sbin/alternatives --install /usr/bin/javac javac /usr/java/jdk1.8.0_161/bin/javac 2000
```

Linux 'alternatives' utility

This utility allows you to use different versions of applications in your environment. By default it is installed in most Linux distributions. If it is not installed, install it via the normal means for installing packages for your Linux distribution.

Ant 1.9.x

Execute the following command to install Ant:

```
$ sudo yum install ant
```

Note: The default installation version for Ant may be different than 1.9 for your system. Check the package version in yum before installing. This prerequisite is only required to build sample UoCs for Java. You can exclude the Java UoCs when building sample UoCs if desired.

Qt 5.2.1

Download qt-opensource-linux-x64-5.2.1.run from location https://download.qt.io/archive/qt/5.2/5.2.1/

Execute the following commands to start the installer:

```
$ chmod 777 qt-opensource-linux-x64-5.2.1.run
```

```
$ ./qt-opensource-linux-x64-5.2.1.run
```

When prompted by the installer enter "/opt/Qt5.2.1" as the install directory and install the program

Append Qt to your path by executing the following command:

```
$ export PATH=$PATH:/opt/Qt5.2.1/5.2.1/gcc_64/bin
```

To avoid repeating these in each terminal window, it is recommended to add this line to the end of your user's bashrc script file located at ~/.bashrc . This will cause the path to be updated in each terminal window.

System Requirements – Windows

The installer for Windows includes checks for the required prerequisites needed to run the CTS application. If a prerequisite isn't found the user will have the option of continuing with the install but is warned that problems may arise with CTS if dependencies are not able to be located.

Below is an overview of the prerequisites needed. Please carefully follow the instructions for installing each.

Overview

Common requirements:

- Python version 2.7.x (not 3.x) with zlib support and setuptools
- Java JDK 7 (also known as 1.7)
- a PDF Viewer (Windows 10 should include a default viewer using Edge, otherwise please obtain one if your system has no PDF viewer)
- Recommended 4GB+ RAM

Language-specific requirements:

C/C++/Ada:

- msys 2.0 with packages mingw-w64-x86_64-toolchain, base-devel, msys2-devel, make

Java:

- Java Development Kit (JDK) 8
- Ant 1.9.x
- QT 5.2.1

Detailed Instructions for Installing Prerequisites

Java JDK 7

Download or acquire JDK 7 from Oracle for Windows 64 bit and install:

http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html

Create a SYSTEM level environment variable JDK7_HOME set to the folder where you installed JDK 7 (example: C:\Program Files\Java\jdk1.7.0_80).

Create a SYSTEM level environment variable JAVA_HOME set to the value: %JDK7_HOME%
The reason for this is when using the Java sample code or otherwise using Java for FACE, JDK 8 is required. This allows both to exist together and the user to switch between them with the JAVA_HOME variable.

Add %JAVA_HOME%\bin to your SYSTEM environment PATH variable at the top of the list.

*****IMPORTANT*****
If installing multiple versions of Java, after completing the final installation you must go to C:\Windows\System32 and rename or delete the copy of java.exe that the installer puts there. Otherwise, the Path variables defined above will NOT work appropriately to switch Java versions since the C:\Windows\System32 version will always be found first.

Python 2.7

Download or acquire and install Python 2.7.x 64 bit for Windows

https://www.python.org/downloads/release/python-2715/

Add the Python installation folder to your SYSTEM environment path variable at the top of the list (ex C:\Python27)

msys 2.0 (for C/C++/Ada samples only)

Download or acquire msys 2.0 from:

https://www.msys2.org/

Install to C:\msys64

After installing msys 2.0, open a msys terminal and install several additional packages by typing the following and pressing enter:

```
pacman -S mingw-w64-x86_64-toolchain base-devel msys2-devel make
```

Select the default - install all (Pressing Enter multiple times)

Confirm with Y for yes

Open file C:\msys64\msys2_shell.cmd and edit line

“rem set MSYS2_PATH_TYPE=inherit” by removing 'rem' which will look like the following when done:

“set MSYS2_PATH_TYPE=inherit”

Add the following to your SYSTEM environment path variable near the top

C:\msys64\mingw64\bin

C:\msys64\usr\bin

Java JDK 8 (for Java samples only)

Download or acquire JDK 8 from Oracle for Windows 64 bit and install:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Create a SYSTEM level environment variable JDK8_HOME set to the folder where you installed JDK 8 (example: C:\Program Files\Java\jdk1.8.0_151).

*****IMPORTANT*****
After installing multiple versions of Java, after completing the final installation you must go to C:\Windows\System32 and rename or delete the copy of java.exe that the installer puts there. Otherwise, the Path variables defined above will NOT work appropriately to switch Java versions since the C:\Windows\System32 version will always be found first.

When you are building the Java samples, set your SYSTEM level environment variable JAVA_HOME set to the value:
%JDK8_HOME%
Revert it to %JDK7_HOME% afterwards.

Ant 1.9.x

Obtain the apache-ant-1.9.9-bin.zip which is included with the ExtInstallers.zip package as part of the release of CTS.

Extract the apache-ant-1.9.9-bin.zip to C:\Program Files\

Create a SYSTEM level environment variable ANT_HOME set to the folder where you installed Ant 1.9.9 (example: C:\Program Files\Java\apache-ant-1.9.9).

Add the following to your SYSTEM environment path variable near the top
%ANT_HOME%
%ANT_HOME%\bin

Note: This prerequisite is only required to build sample UoCs for Java. You can exclude the Java UoCs when building sample UoCs if desired.

Qt 5.2.1

Obtain the Qt 5.2.1 installer which is included with the ExtInstallers.zip package as part of the release of CTS.

Run the installer and select all default settings.

Note: This prerequisite is only required for Java.

Installation - Linux

To install the Conformance Test Suite (CTS), simply extract the archive file (zip or tar.gz) to a folder somewhere under your user's home folder where you have read/write/executable access.
You must also setup Java environment variables. First ensure the environment variable JDK7_HOME is defined pointing to the base directory of the JDK 7 installation. It is recommended to add this to your permanent environment or in a startup script like ~./.bashrc. It must be defined and exported.
Example:

```
$ export JDK7_HOME=/usr/java/jdk1.7.0_80
```

If the Java sample UoCs will be used, the JDK8_HOME variable must also be defined to point to the JDK 8 base directory.
Example:

```
$ export JDK7_HOME=/usr/java/jdk1.8.0_121
```

Set your JAVA_HOME variable to the JDK7_HOME variable anytime you need to run the CTS GUI.
Example:

```
$ export JAVA_HOME=$JDK7_HOME
```

Building the Sample Projects and Toolchains

You can optionally build the sample projects and generate toolchain files using an included python script. These must be generated using this script as they use an absolute path relative to the folder where you have extracted the CTS. To generate these files follow these steps.

Set the JAVA_HOME variable to JDK 8 in order to be able to build the Java sample projects.
Example:

```
$ export JAVA_HOME=$JDK8_HOME
```

(Note this is not required if only the C/C++/Ada samples will be generated).

Using a terminal, navigate to the top folder of the CTS.

Execute the following commands.

```
$ cd sample
```

If you are using all languages (C, C++, Ada, and Java) you can use this command to build all samples and generated files for the samples:

```
$ python testUtility.py --gen_only
```

If you are using only C/C++ and profiles General/Security you can use this command to build C/C++ samples and generated files for the profiles General/Security samples:

```
$ python testUtility.py --gen_only -cp --general --security
```

Note that most commands can be limited to language or profile with this script with the appropriate switches:

- -c for C, -p for C++, -a for Ada, and -j for Java
- --general for General, --safety_base for Safety Base, --safety_ext for Safety Ext, and --security for Security

Sample projects and code are provided for each FACE segment. For the TSS and IOS segments, the UoC name is assumed to be "UOPName." For the Portable Components Segment (PCS), the UoP name is taken from the sample data model and is set to "UoP1". For the Platform Specific Services Segment (PSSS), the UoP name is "UoP2" (also per the sample data model).

Folders under the 'sample' directory are as follows:

- projects - contains sample projects for all languages. Source code for C/C++/Ada is stored alongside the project.
- toolchains - sample toolchains.
- datamodels - sample data model used by sample projects.

Installation - Windows

To install the Conformance Test Suite (CTS), simply run the CTS installer as an administrator. The installer for CTS will ask for the location for where you would like to install. It is recommended to use the default location:

C:\CTS\conformancetestsuite

Do not use a directory under C:\Users\ or on the desktop. Windows path length limits will cause errors with the CTS.

The installation process for CTS contains tasks that will check the system for the availability of all prerequisites needed to run the CTS and allows for install of missing prerequisites. The installer also includes an option to add a desktop shortcut icon to allow for easy start up of the application. Note that the installer will add all environment variables to the system if they were not already defined prior to the CTS install.

Building the Sample Projects and Toolchains

To generate the samples that come with CTS you may use the included python script with CTS. The samples to be generated are the sample projects and the sample toolchains. You must use the script as it uses an absolute path relative to the folder where you have extracted the CTS. To generate these files:

1. Open a Windows command prompt
2. Open a mingw64 shell using command:
   C:\msys64\msys2_shell.cmd -mingw64
   navigate to the "sample" directory of the CTS.
   Example Command: cd /c/CTS/conformancetestsuite/sample
3. Run the test utility script to generate/build the projects
   python testUtility.py --gen_only -w
4. Keep the msys2 terminal open for further commands

Samples

After running the testUtility script, sample project files (PCFGs) will be generated under the sample/projects/<lang>/<segment>/<profile> subdirectories.

Sample toolchain configuration files (TCFGs) will be generated under the sample/toolchains/<lang>/<profile> subdirectories.

This will also build sample UoC files and OSS gold standard libraries which are referenced by sample projects.

The testUtility.py script must be run with the --gen_only flag prior to using the script to run tests on the sample projects or errors involving missing generated files may occur. This will attempt to generate/build projects and UoCs for all languages. If desired, specific languages can be specified with additional flags. To display all flags use the following:

```
$ python testUtility.py -h
```

For example, to generate and build UoCs for only C and C++ use:

```
$ python testUtility.py --gen_only -c -p
```

Note 1: The testUtility script generates project files for the CTS (with the PCFG file extension) from templates. These templates are not native CTS projects. They are used only for the sample projects since the project file requires an absolute path as the base directory for the project. The testUtility generates the project files using your system's path to the CTS. The template files (*.pcfgtemplate) are not complete CTS files and cannot be opened with the CTS GUI.

Note 2: The samples provided are configured for a GCC / GNAT based toolchain. To use a different toolchain, Modify tcfgtemplate file for the desired language with a text editor. Then rerun testUtility.py with -e -l flags to regenerate the toolchain configurations.

Note 3: There are some sample OSS projects that are included with Linux but are not included with the Windows distribution. This difference is due to some of the sample OSS projects (C and C++) for Windows fails due to mingw not being FACE conformant.

To automatically run all the samples from command line use the following:

```
$ python testUtility.py -r
```

To limit the projects run to a specific language use:

```
$ python testUtility.py -r [language flag]
```

Example (for C++ and Security profile):

```
$ python testUtility.py -r -p --security
```

Running the Conformance Test Suite - Linux

The current version of Java in the PATH must be set to Java 7 before running the CTS

Ensure that this is setup by running the following command:

```
$ java -version
```

Check this is a Java 7 version. Note that the OpenJDK is not supported as it does not provide JavaFX which the CTS GUI uses. If this is not the currently selected version of java, use the alternatives command to switch:

```
$ sudo alternatives --config java
```

When promoted enter the number corresponding to the JDK 1.7

To start the CTS, navigate to the CTS root directory and run the CTS GUI by executing the following command:

```
$ ./run_CTS_GUI.py
```

Using both Java 7 and Java 8 - Linux

The sample CTS projects for Java use Java 8. To use both Java 7 and Java 8 in the same environment, install the Linux utility called 'alternatives' as described in the System Requirements section. If using both Java 7 and Java 8 in the environment and the alternatives utility is installed, you can switch your current Java version to 7 using the following command:

```
$ sudo alternatives --config java
```

When prompted enter the number corresponding to the JDK 1.7.

Note that the OpenJDK **is not supported as it does not provide JavaFX which the CTS GUI uses JavaFX.**

Next, ensure that javac is set to use the Java 8 by executing the following commands:

```
$ sudo alternatives --config javac
```

When prompted enter the number corresponding to the JDK 1.8.

Running the Conformance Test Suite - Windows

All instructions from the System Requirements section must be performed first.

If a desktop icon was added to the desktop per the installer, simply double click the Conformance Test Suite shortcut icon to start the application. Otherwise follow the instructions below to start the application manually:

1. Open a Windows command prompt
2. Check that the current Java version is Java 7 using the following command (note the $ is just to indicate a terminal, do not type this):
   ```
   $ java -version
   ```
   If it is not Java 7, set your system environment variable JAVA_HOME to %JDK7_HOME% (refer back to the System Requirements section for Windows to ensure your environment variable and PATH are set correctly)
3. Navigate to the directory where you extracted the CTS
4. Run the following command:
   ```
   $ python run_CTS_gui.py
   ```

Using both Java 7 and Java 8 - Windows

The sample CTS projects for Java use Java 8. To use both Java 7 and Java 8 in the same environment, you can switch your current Java version using the system environment variable JAVA_HOME you setup in the System Requirements section. Note that you should not have to switch between the Java versions, but if you need to please follow the aforementioned instructions.

Note Regarding Failing Test Results and Shared Data Model

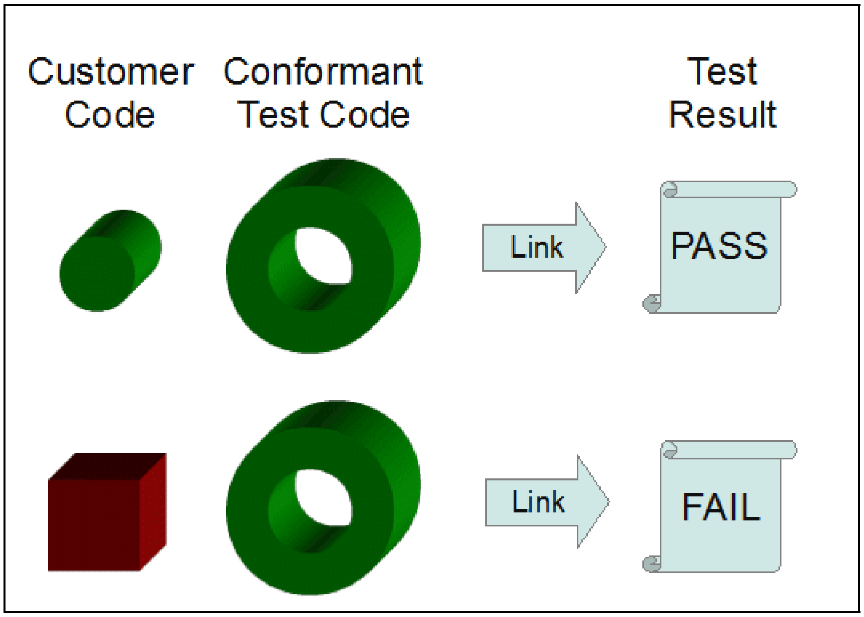
Part of the full conformance test is a test of the data model provided by the project, where applicable. Part of the data model test involves testing of the Shared Data Model. As of this release of the CTS, no Shared Data Model exists for FACE 3.0. Therefore, for all sample projects, the USM data model is used for both the USM data model and Shared Data Model (SDM). Because of this, the SDM portion of the data model test will fail. Since the data model test fails, the overall test result is marked as failed in the test report and the CTS. However, if you examine the report you can that see the rest of the test results are shown separately as PASS.

For the sample projects, the expected result is PASS for all sample projects except for the C_OSS_POSIX.pcfg and CPP_OSS_CPP03.pcfg tests on Linux and all of the C and C++ OSS tests on Windows.(This is due to the fact that both Linux and Windows are not FACE Conformant).

Theory of Operation

For C, C++, and Ada code, conformance is determined by mating customer code with corresponding conformant test code. Customer applications will be linked with FACE test interfaces. Customer interface libraries will be linked against by FACE test applications. The test interfaces provide all possible function calls, data types, and constants available to the customer code. The test applications utilize all possible function calls, data types, and constants that should exist in the customer code. The test applications are compiled using the customer's header files or spec files (for C/C++/Ada) and then linked against both the customer's code and the test libraries that contain the function calls, data types, and constants available to the customer code. If the compile and link pass, the customer code is conformant with respect to the requirements tested. If the compile or link fail, the customer code is not conformant. Errors are included in the test output.

The test only determines conformance with respect to function signature. The test neither proves nor disproves correctness of functionality or correctness of function usage. Additionally, for testing the existence of abstract interfaces, the test does not determine if the customer code actually implements the interface, only that the abstract interface is defined correctly in the customer's headers or spec files. For testing existence of non-abstract interfaces, the test determines if the interface is defined in the customer code. For testing use of non-abstract interfaces, the test determines if the interface used by the customer's code is an allowed interface and will only pass if that interface is allowed for the UoC under test.



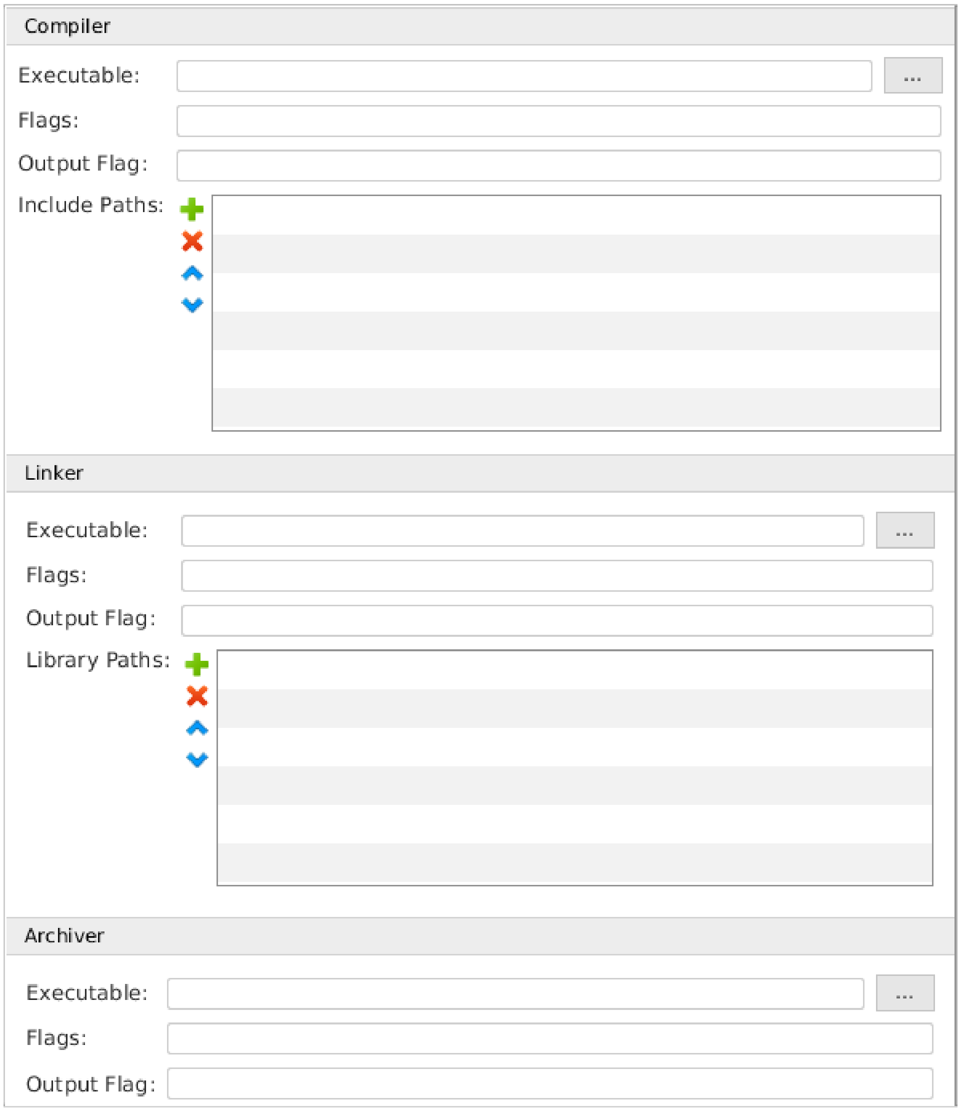
Introduction to Methodology

Two methods of performing the link test exist. One uses the target linker. The other uses the host linker. The target linker is the linker used to produce an executable targeting the embedded system. The host linker is the linker used to produce an executable targeting the development system where the CTS runs. Each method has its own advantages.

The target linker method is advantageous in that a project's existing build infrastructure can be reused during conformance testing. Additionally, any conditionally compiled code based on hardware architecture which is reflected in the compiler and linker will be included in the conformance testing. The disadvantage is that conformance testing staff must know the details of the target linker.

The host linker is advantageous in that its usage details are preselected in the conformance tool. Its disadvantage is that conditionally compiled code based on hardware architecture which is reflected in the compiler and linker may not be included in conformance testing. Additionally, the project's build infrastructure would need to be modified to make use of the host compiler and linker.

Target Linker Method



If you choose the target linker method, you must provide the conformance tool details about your build tools. You must provide the path to and name of the compiler, linker, and archiver for your build tools. Additionally, you must provide compiler flags, linker flags, and archiver flags to provide correct behavior. The flags must instruct the tools to ignore any system included code such as standard headers and libraries. The flags must also select the correct target language standard. The table below provides the minimum set of equivalences you must provide.

Flag Purpose		*GNU Tools Example*
Language standard selection		
C	ISO C 1999	
C++	ISO C++ 2003	-std=c99
Ada	ISO Ada 1995	-std=c++03 (or c++0x on some compilers)
		-std=-gnat95

For Non OSS Tests:

(compiler flags)

Disable bundled headers	-nostdinc (-nostdinc++)
Disable builtin functions	-fno-builtin

(linker flags)

Disable builtin libraries	-nodefaultlibs -nostartfiles

Host Linker Method

If you choose the host linker method, you must alter your project's build system to use the host's build tools and recompile. Be mindful of any conditionally compiled code based on architecture or compiler.

Additional Methodology Information

When building your project, alter your compiler flags to include the conformance tool's goldStandardLibraries directory for IOS, TSS, and OSS headers. This gives you a known good starting point so that your code is tested rather than the code provided with your build tools which could result in false positives. Details on how to achieve this is described above in the Target Linker Method section.

OSS Testing Methodology

Unlike the other segments, to test the OSS for FACE conformance, the system libraries and include files will need to be used. You will want to specify the language standard, but you will not want to disable the headers and built in functions and libraries. You will also need to specify the location of include files and libraries to be used in the system test, either by compiler and linker option flags, or by selecting include paths and libraries via the configuration GUI as described in the Testing an Operating System (OSS) Segment section below.

Java Testing Methodology

The Java testing methodology differs greatly from the methodology for C, C++, and Ada. This is due to the standardized data format of Java's .class files allowing these files to be universally queried for information.

PCS and PSS segment class files are queried for their dependencies such as any classes, methods, or fields necessary to execute. These dependencies are compared against a white list as defined by the standard. Violations are reported as errors.

OSS, TSS, and IOS segment class files are queried for their capabilities such as classes, methods, and fields as well as attributes for each. These are compared against a minimum list as defined by the standard. Any omissions or incorrect definitions are reported as errors. Additionally, native methods are flagged as warnings to inspect.

Initializing the Conformance Test Suite

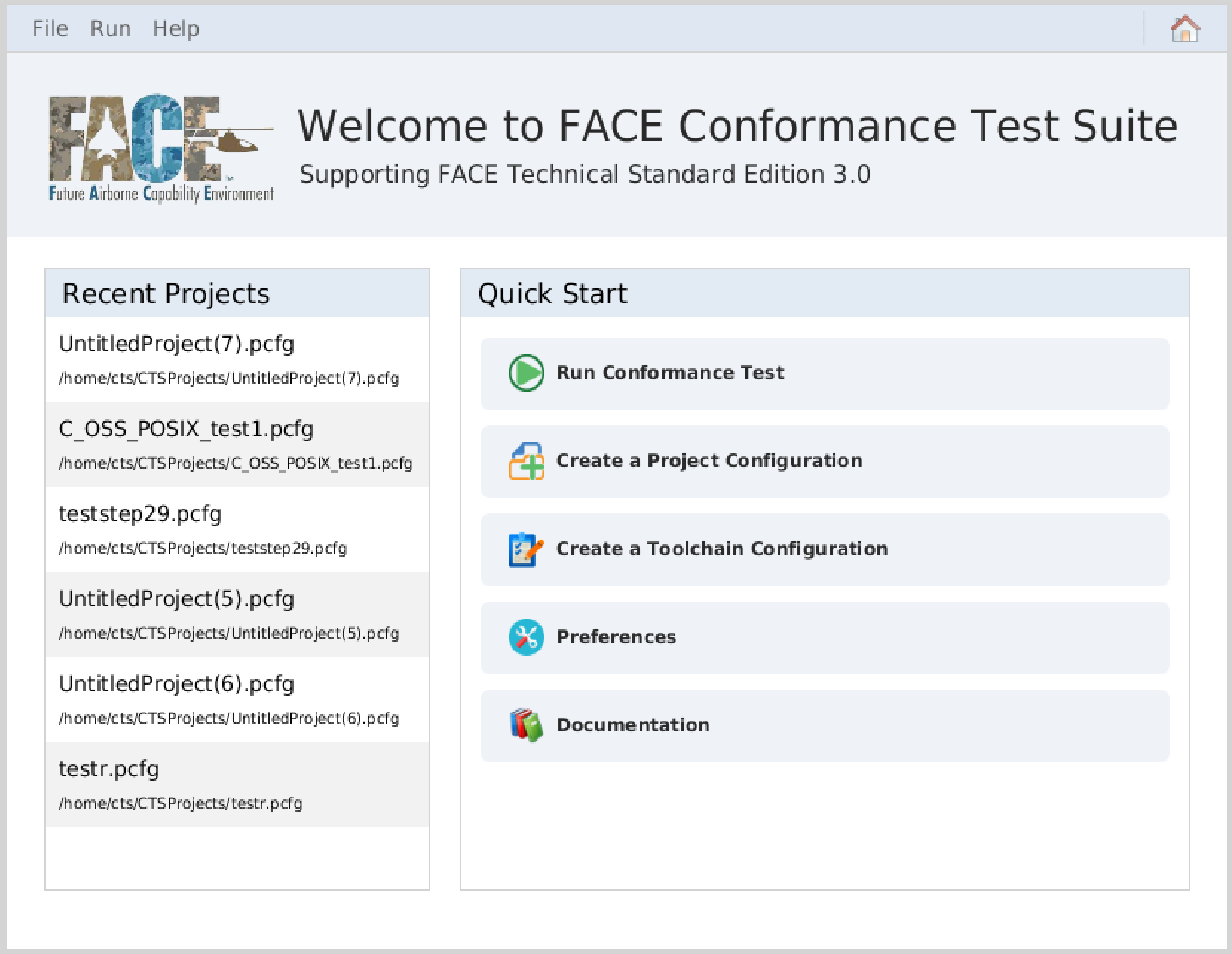
Procedure

Ensure the JDK7_HOME environment variable has been defined per the installation instructions in the previous sections.

Start the conformance test suite by running the run_CTS_GUI.py script in the main test suite directory from the command line.

On Windows, you may instead use the shortcut created by the installer (if installation was performed via the installer and not manually).

This will launch the conformance main menu as shown below:


File
Run
Help
FACE
Future Airborne Capability Environment
Quick Start
Run Conformance Test
Create a Project Configuration
Create a Toolchain Configuration
Preferences
Documentation

Welcome to FACE Conformance Test Suite

Supporting FACE Technical Standard Edition 3.0

Recent Projects

UntitledProject(7).pcfg
/home/cts/CTSProjects/UntitledProject(7).pcfg

C_OSS_POSIX_test1.pcfg
/home/cts/CTSProjects/C_OSS_POSIX_test1.pcfg

teststep29.pcfg
/home/cts/CTSProjects/teststep29.pcfg

UntitledProject(5).pcfg
/home/cts/CTSProjects/UntitledProject(5).pcfg

UntitledProject(6).pcfg
/home/cts/CTSProjects/UntitledProject(6).pcfg

testr.pcfg
/home/cts/CTSProjects/testr.pcfg

A high level overview of the steps a user will follow to use the Conformance Test Suite is as follows:

1. Create the data model for your UoC (if your UoC uses or provides any type-specific interface – the TSS Type Specific or Life Cycle Management (LCM) Stateful interfaces).
2. Create a toolchain for your specific compiler/linker/archiver tools, either from scratch or basing it off one of the sample

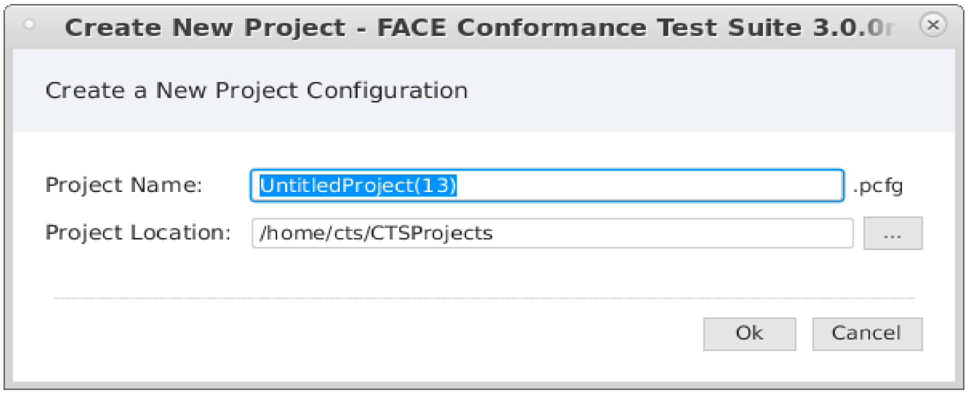
toolchains. The sample toolchains can also be used directly if desired.

3. Create a basic project by specifying the profile, segment, interfaces the UoC implements and interfaces it uses, and the data model (if appropriate).
4. Run the “Generate GSLs/Interface” button in the toolbar. This will generate all the interface headers (C/C++) / Ada spec files / Java files for the interface that UoC can access or will implement. These files will be placed into a subfolder of the folder specified in the project as the “Gold Standard” folder (the relative path of the subfolder is include/FACE).

   This process also generates a text file in this location with all the include paths the user should use to compile their code for conformance.
5. The user writes their implementation code that implements each interface being provided by the UoC from these generated interfaces created in Step 3. For example, for C++ and Java, the implementation is a derived class for each interface being provided. The base/abstract class is the interface class provided in the Gold Standard subfolder include/FACE as generated by the CTS. For C and Ada, one must create implementations of the functions/procedures. Next, for each FACE interface that the UoC is to “use” (access), the user must also implement the Injectable interface for that interface.
6. User compiles their UoC code using the generated headers or spec files or Java files (depending on language) and the include paths (compiler paths or class paths) provided in the generated text file.
7. User completes their CTS project pointing it to the implementation object files (or classes for Java) and runs the CTS.

These steps will be detailed in the following sections.

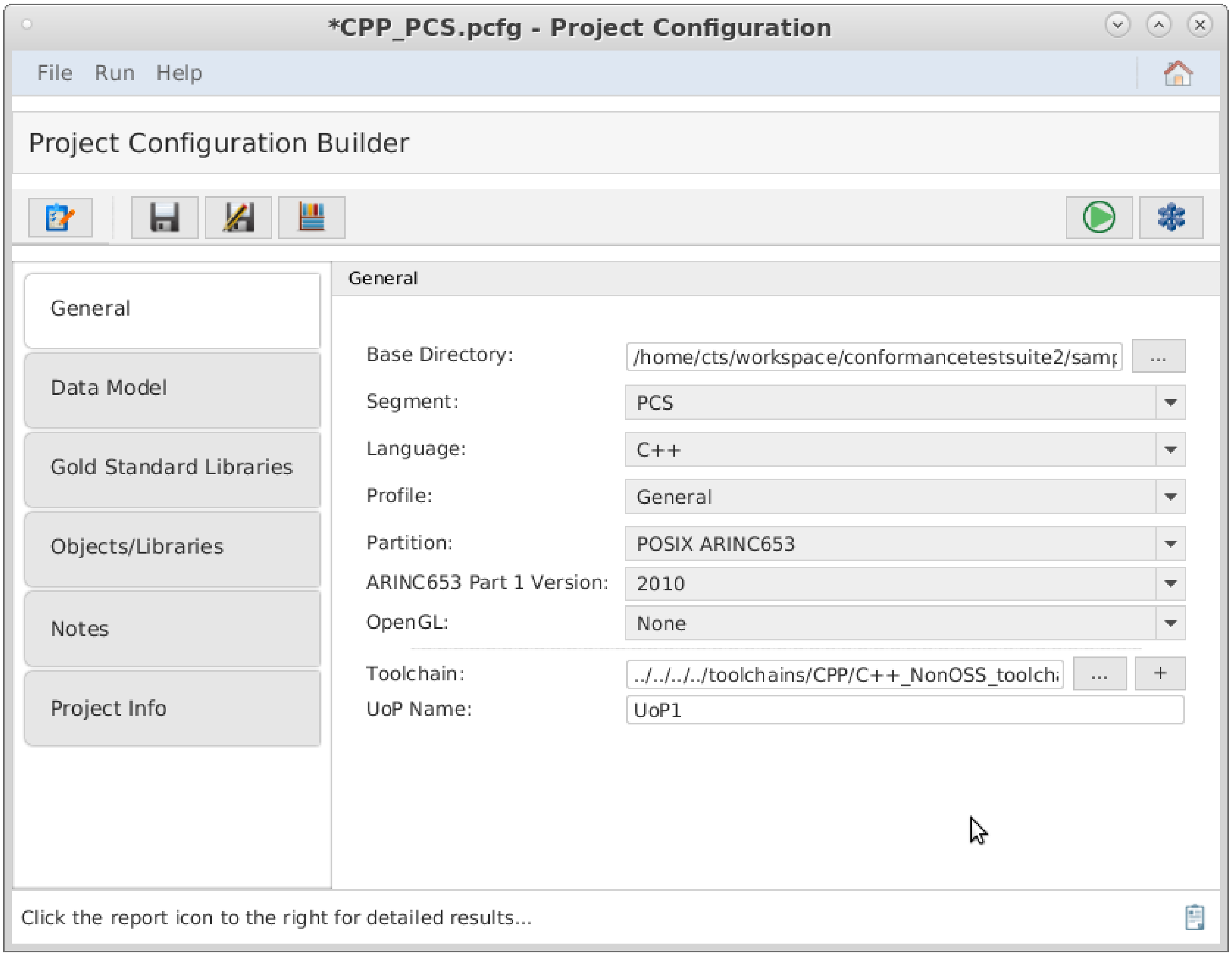
1. Click the “Create a Project Configuration” button to launch the display below.


Create New Project - FACE Conformance Test Suite 3.0.0
Create a New Project Configuration
...
Ok
Cancel

Project Name: UntitledProject(13) .pcfg

Project Location: /home/cts/CTSProjects

2. Enter a name for the project. The default project location will be ‹home›/‹user›/CTSProjects. Press 'Ok' to launch project configuration builder shown below.



*CPP_PCS.pcfg - Project Configuration

Project Configuration Builder

General

Base Directory: /home/cts/workspace/conformancetestsuite2/samp

Segment: PCS

Language: C++

Profile: General

Partition: POSIX ARINC653

ARINC653 Part 1 Version: 2010

OpenGL: None

Toolchain: ../../../toolchains/CPP/C++_NonOSS_toolch

UoP Name: UoP1

Click the report icon to the right for detailed results...

3. Select the “General” tab at the top of the project configuration builder to display the general information options as shown above.

4. Select the FACE architecture segment of the UoC (PCS, PSSS, TSS, IOSS, OSS).

5. Select the programming language used by the segment(s) interface under test.

6. Select the FACE profile to be used by the segment(s) under test.

7. Select the operating system partition type to be used by the segment(s) under test.

   ARINC 653: For an OSS UoC providing ARINC 653 APIs, this indicates the UoC provides all the required ARINC 653 APIs for the selected profile in the FACE Technical Standard 3.0 (further inputs are required in the Objects/Libraries tab as described later). For all other UoCs, this indicates the UoC may use any of those ARINC 653 APIs provided by another OSS UoC as required by the standard.

   POSIX: For an OSS UoC providing POSIX APIs, this indicates the UoC provides all the required POSIX APIs for the selected profile in the FACE Technical Standard 3.0 (further inputs are required in the Objects/Libraries tab as described later). For all other UoCs, this indicates the UoC may use any of those POSIX APIs provided by another OSS UoC as required by the standard.

   POSIX ARINC 653: For an OSS UoC providing POSIX and ARINC 653 APIs, this indicates the UoC provides all the required POSIX APIs for the selected profile in the FACE Technical Standard 3.0, and the subset of required ARINC 653 APIs that an OSS UoC in a POSIX environment can provide as defined in the FACE Technical Standard 3.0 (further inputs are required in the Objects/Libraries tab as described later). For all other UoCs, this indicates the UoC may use any of those POSIX APIs or the subset of ARINC 653 APIs provided by another OSS UoC in a POSIX environment as required by the standard.

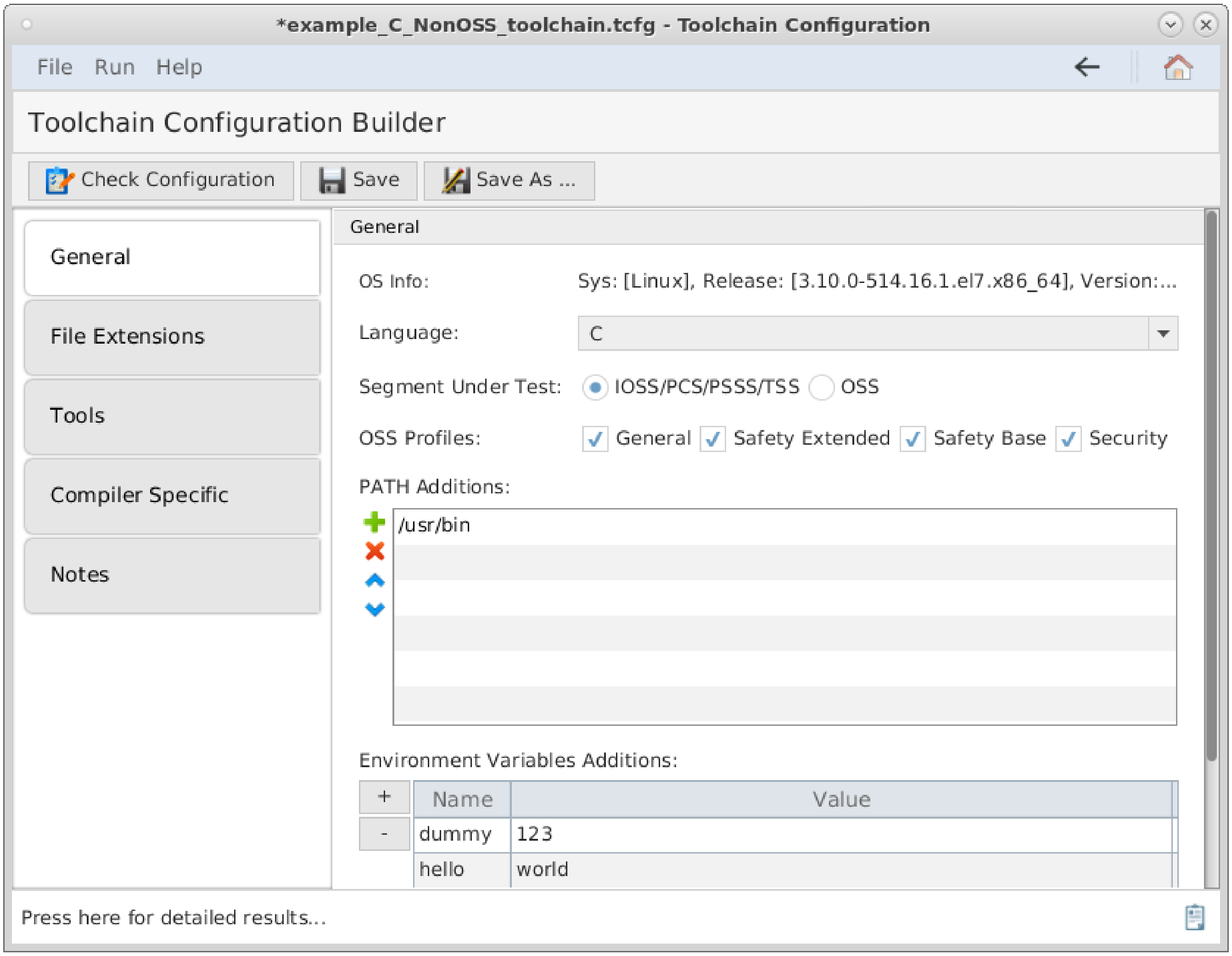
8. If OpenGL is used by a UoC or provided by an OSS UoC, select the version of OpenGL.
9. Select the Toolchain file that will be used for the segment under test. You may either use an existing or sample toolchain file, or create a toolchain configuration by following the procedure in the ***Building a Toolchain Configuration*** section if a toolchain file needs to be created for the project.
10. Set the Unit of Portability (UoP) name for the UoP you will be testing. For PCS and PSSS UoPs (also known as UoCs), this name much match the name given to it in the data model you will be using.

Notes:

By pressing the "..." button, a directory or file dialog box will be launched to allow you to graphically pick your response(s). Pressing the red x will clear the text entry box.

Building a Toolchain Configuration file

1. To begin, click the "Create a Toolchain Configuration" button on the home page of the GUI.


*example_C_NonOSS_toolchain.tcfg - Toolchain Configuration
File
Run
Help
Toolchain Configuration Builder
Check Configuration
Save
Save As ...
General
File Extensions
Tools
Compiler Specific
Notes
General
Press here for detailed results...

OS Info: Sys: [Linux], Release: [3.10.0-514.16.1.el7.x86_64], Version:...

Language: C

Segment Under Test: ● IOSS/PCS/PSSS/TSS ○ OSS

OSS Profiles: ☑ General ☑ Safety Extended ☑ Safety Base ☑ Security

PATH Additions:

/usr/bin

Environment Variables Additions:

Name	Value
dummy	123
hello	world

2. Select the programming language used by the segment interface under test.

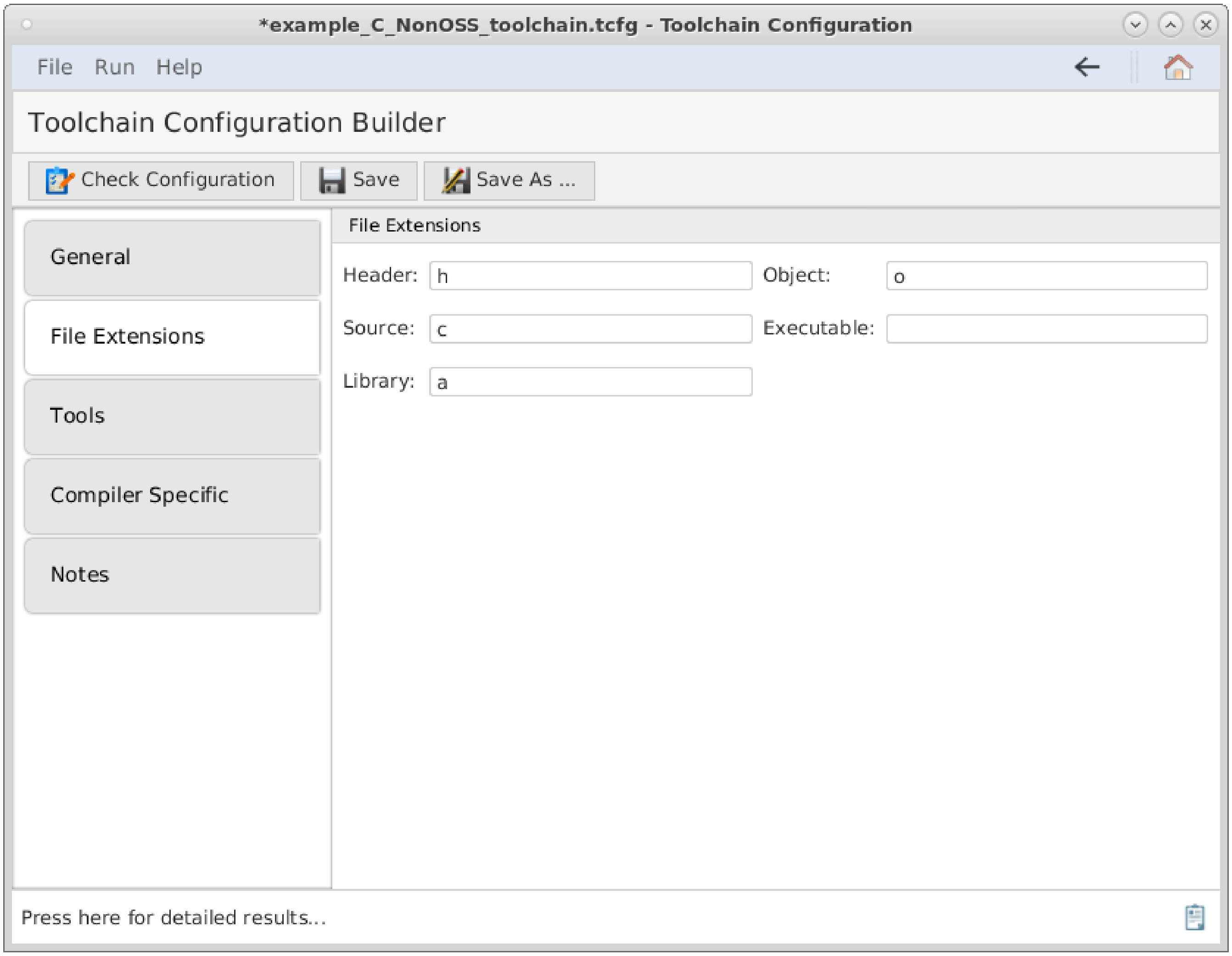
3. Select the type of segment under test.

4. Select the system profile used by segment.

5. Enter any directories to place in the include paths.

6. Add environmental variables.

7. Select the File Extensions tab to display the options below.



*example_C_NonOSS_toolchain.tcfg - Toolchain Configuration

Toolchain Configuration Builder

File Extensions

Header: h

Source: c

Library: a

Object: o

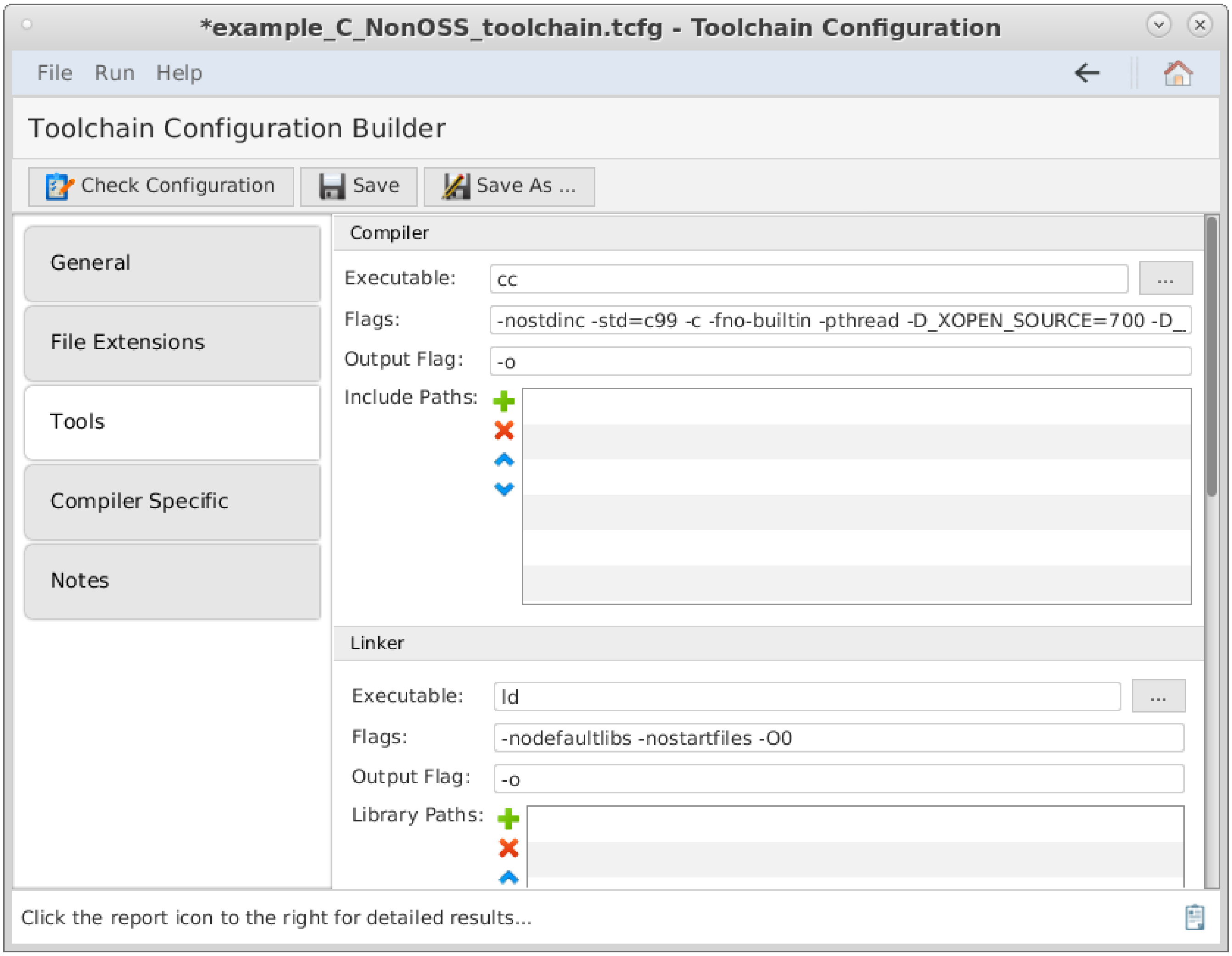
Executable:

8. Choose the extension used for object files and library by the compiler.

9. Choose the extension used for executable files by the compiler. (Leave blank for no extension.)

10. Choose the extension of the header/source files. The extension should be the extension used by the programming language that the segment uses.

11. Select the Tools tab to see all the build options below.



Compiler

Executable: cc

Flags: -nostdinc -std=c99 -c -fno-builtin -pthread -D_XOPEN_SOURCE=700 -D_

Output Flag: -o

Linker

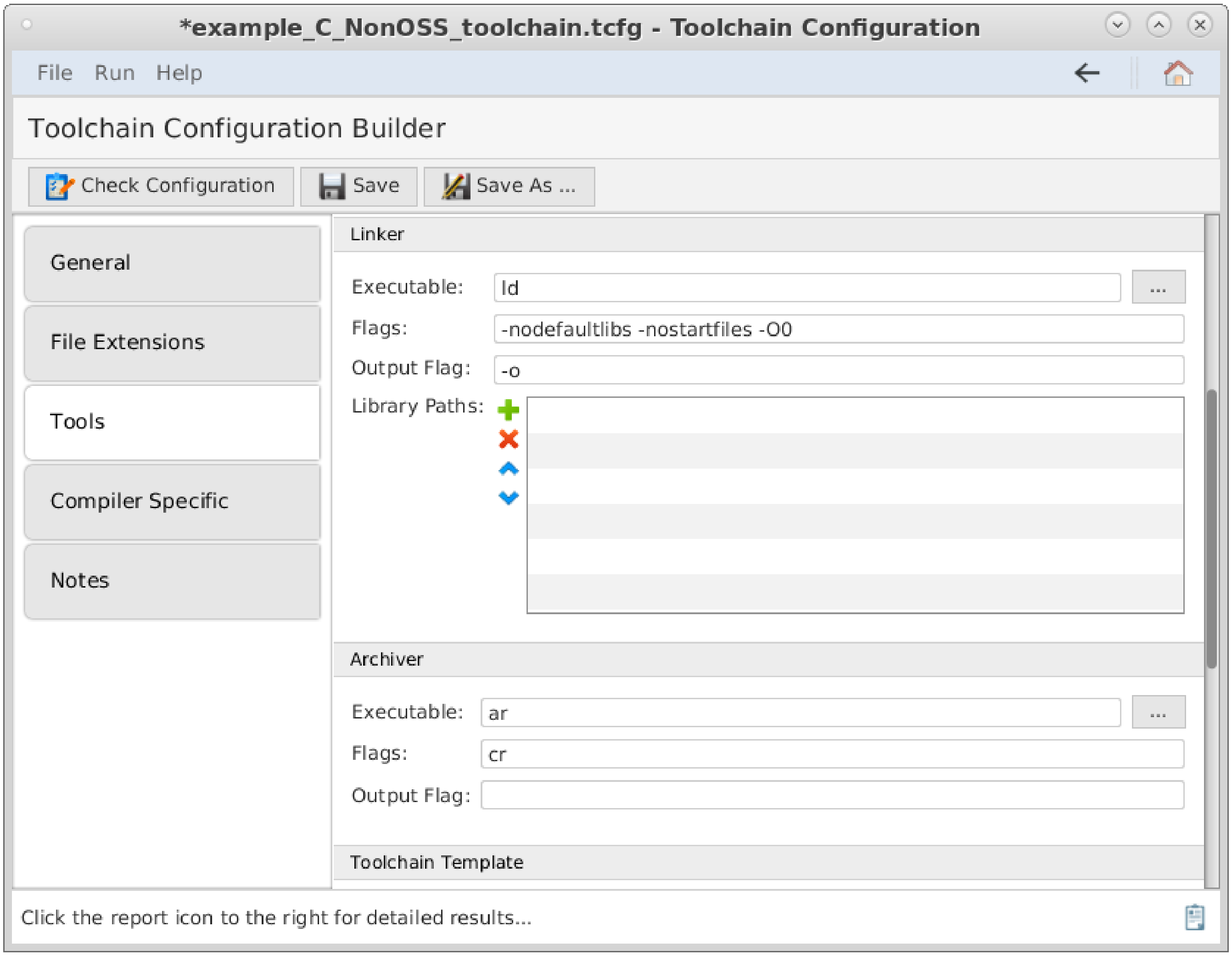
Executable: ld

Flags: -nodefaultlibs -nostartfiles -O0

Output Flag: -o

Click the report icon to the right for detailed results...

12. Choose the build options for compiler, linker, and archiver according to the *Theory of Operation* section above. A field for processor specific compile flags has been provided. For Ada segments, you can choose a binder to use during the build procedure. Note: Please specify the exact compiler, not the compiler collection if possible (i.e. g++ over gcc for C++).**Note: For the sample toolchains, the symbol "FACE_GENERAL_PURPOSE_PROFILE" is defined. This flag is used by the compiler specific "allowed definitions" code (see steps below) and must be set if using these toolchains.**



Linker

Executable: ld

Flags: -nodefaultlibs -nostartfiles -O0

Output Flag: -o

Library Paths:

Archiver

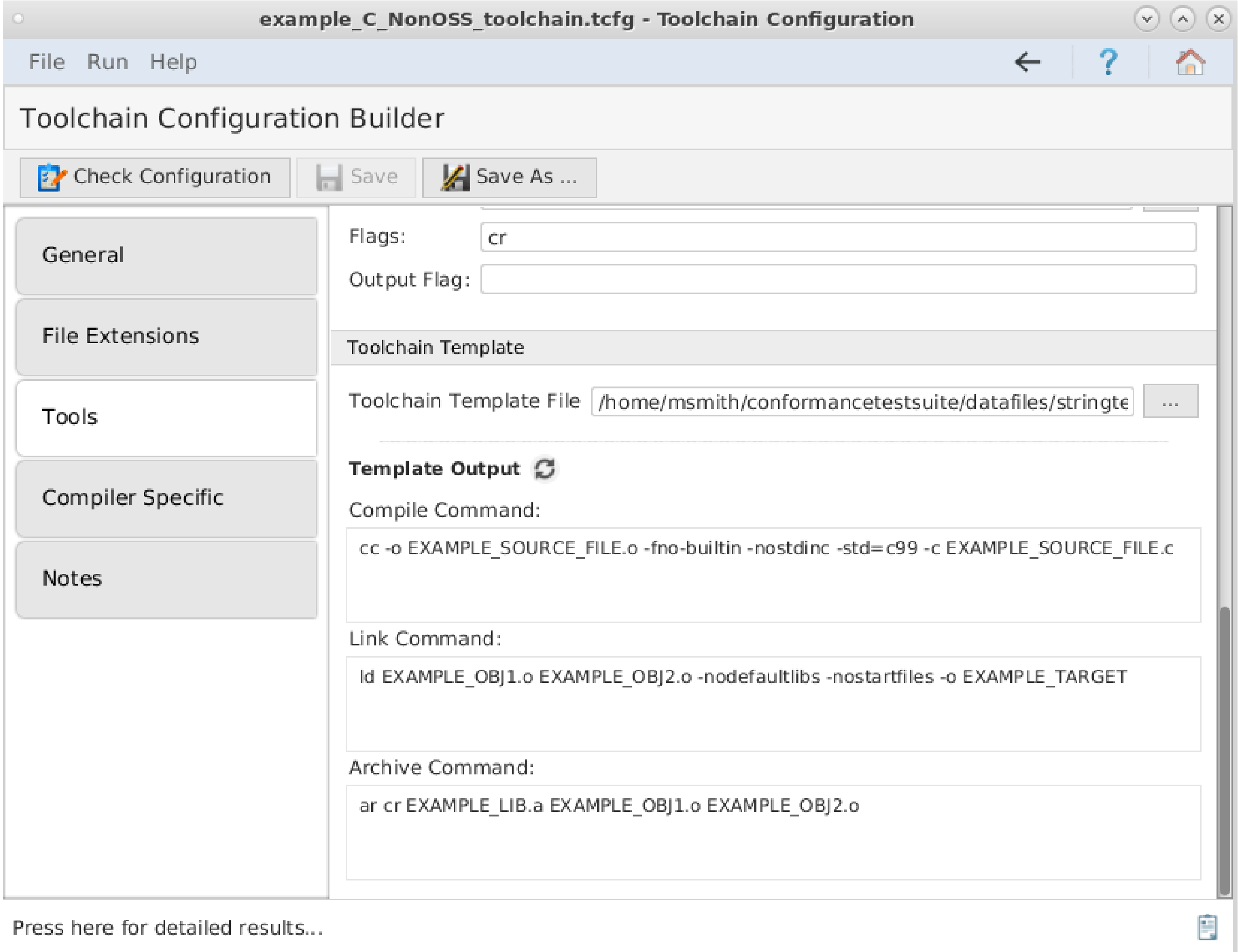
Executable: ar

Flags: cr

Output Flag:

Click the report icon to the right for detailed results...

13. Scroll down to view the toolchain template section shown below.



Flags: cr

Output Flag:

Toolchain Template File /home/msmith/conformancetestsuite/datafiles/stringte

Compile Command:

```
cc -o EXAMPLE_SOURCE_FILE.o -fno-builtin -nostdinc -std=c99 -c EXAMPLE_SOURCE_FILE.c
```

Link Command:

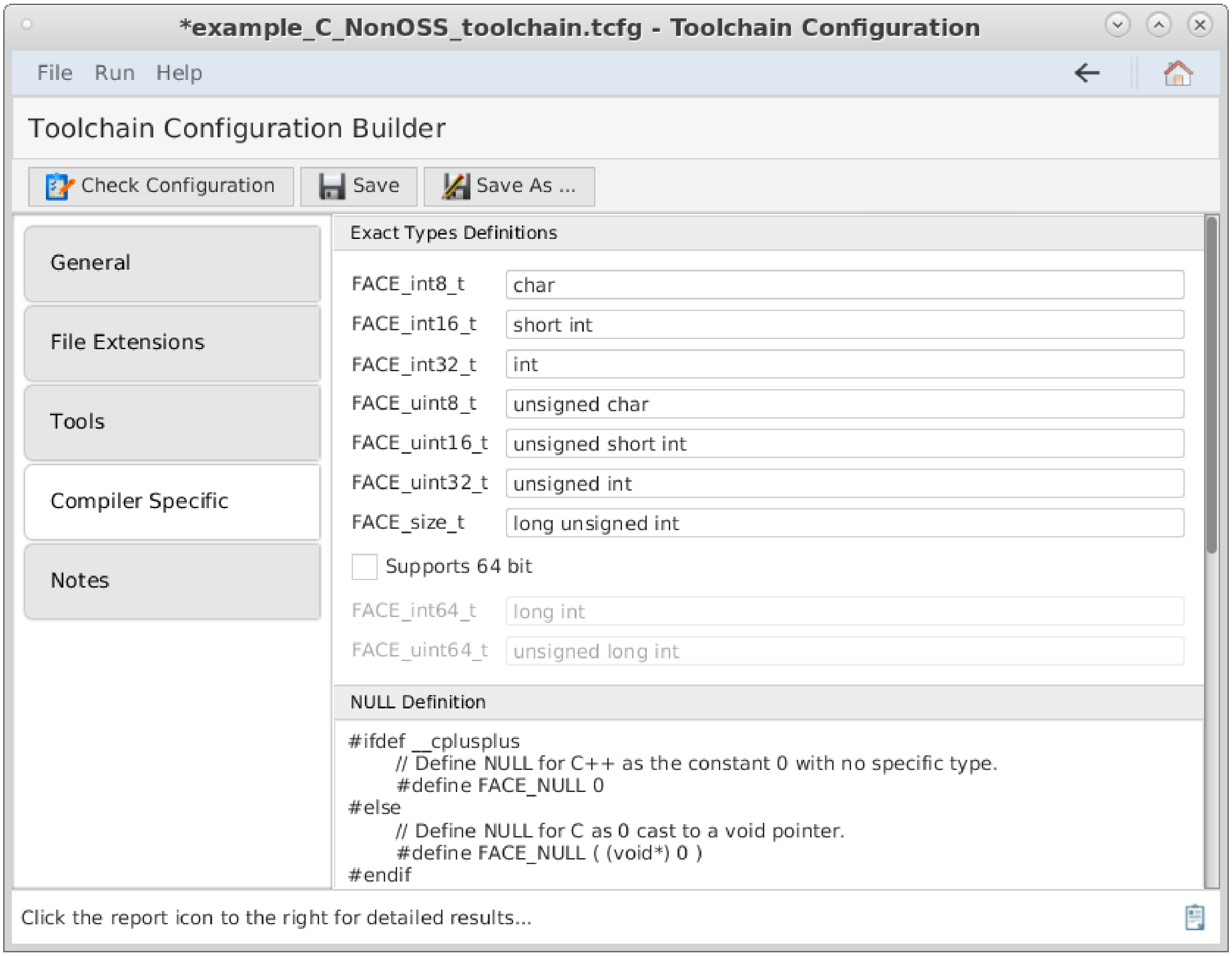
```
ld EXAMPLE_OBJ1.o EXAMPLE_OBJ2.o -nodefaultlibs -nostartfiles -o EXAMPLE_TARGET
```

Archive Command:

```
ar cr EXAMPLE_LIB.a EXAMPLE_OBJ1.o EXAMPLE_OBJ2.o
```

14. Add toolchain template file (stg) by clicking the ... button. Note: Without the toolchain template file the toolchain will be invalid. Toolchain templates of each language can be found in the datafiles/stringtemplate folder of the test suite directory. To view the commands click the '...' button. After selecting the template and defining the various toolchain commands/flags above, you may click the refresh button next to the Template Output header to show the example commands the CTS will use based on the commands you have configured and the selected template.

15. Select the Compiler Specific tab to display the options below.


*example_C_NonOSS_toolchain.tcfg - Toolchain Configuration
File
Run
Help
Toolchain Configuration Builder
Check Configuration
Save
Save As ...
General
File Extensions
Tools
Compiler Specific
Notes
Exact Types Definitions
NULL Definition
Click the report icon to the right for detailed results...

FACE_int8_t	char
FACE_int16_t	short int
FACE_int32_t	int
FACE_uint8_t	unsigned char
FACE_uint16_t	unsigned short int
FACE_uint32_t	unsigned int
FACE_size_t	long unsigned int

☐ Supports 64 bit

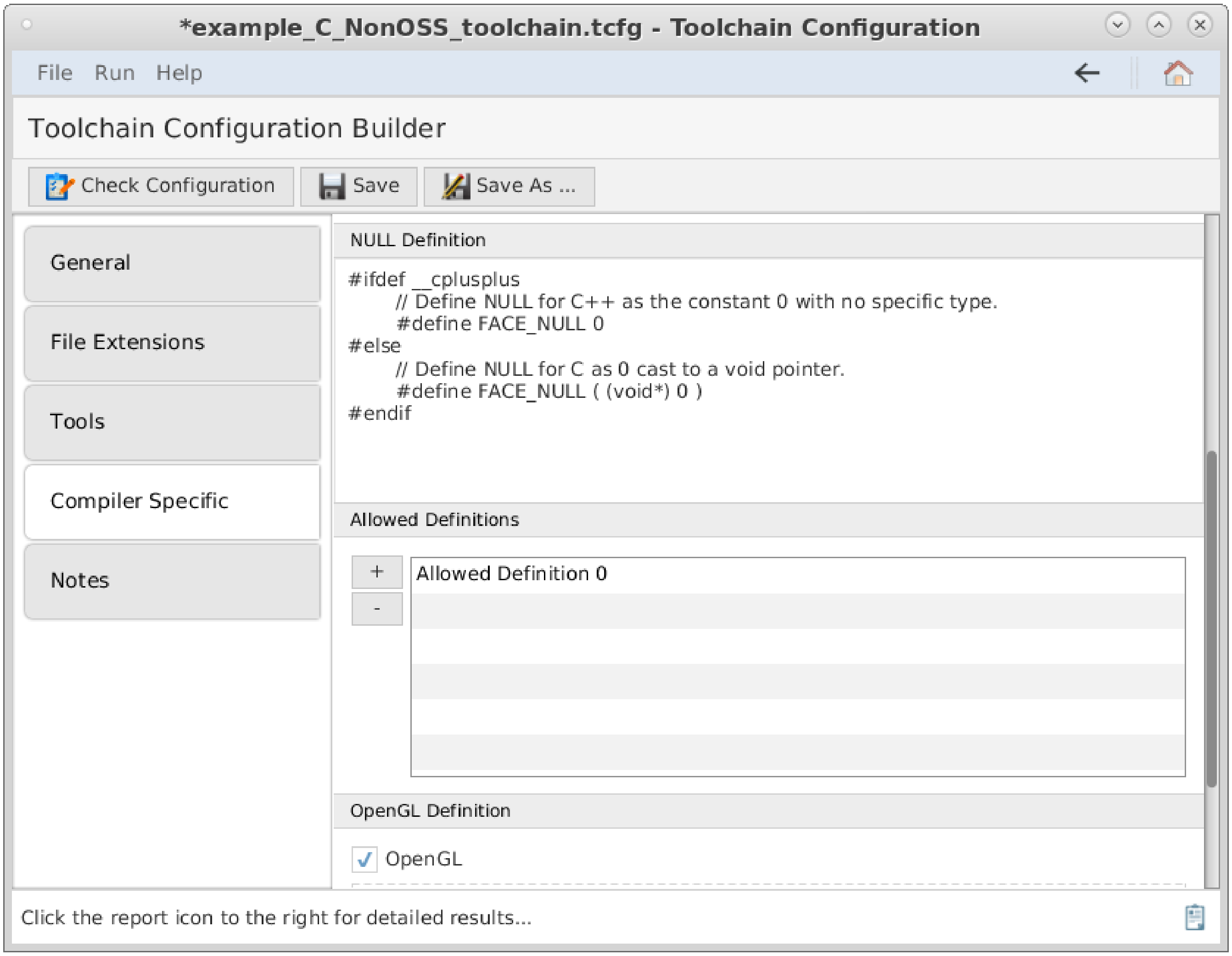
FACE_int64_t	long int
FACE_uint64_t	unsigned long int

```
#ifdef __cplusplus
        // Define NULL for C++ as the constant 0 with no specific type.
        #define FACE_NULL 0
#else
        // Define NULL for C as 0 cast to a void pointer.
        #define FACE_NULL ( (void*) 0 )
#endif
```

16. For C and C++ testing, the exact size types must be configured according to your compiler's manual. This is done through the Compiler Specific section. Consult your compiler's manual to create a typedef mapping between its intrinsic types and the exact types needed for testing.

Testing Type Name	***Description***
FACE_int8_t	8-bit signed integer
FACE_int16_t	16-bit signed integer
FACE_int32_t	32-bit signed integer
FACE_int64_t	64-bit signed integer
FACE_uint8_t	8-bit unsigned integer
FACE_uint16_t	16-bit unsigned integer
FACE_uint32_t	32-bit unsigned integer
FACE_uint64_t	64-bit unsigned integer
FACE_size_t	Unsigned integer type of the result of sizeof()

17. Add the NULL definition. The NULL type must be configured according to your compiler's manual. This is done by entering the null TYPES in the null definition section shown below:



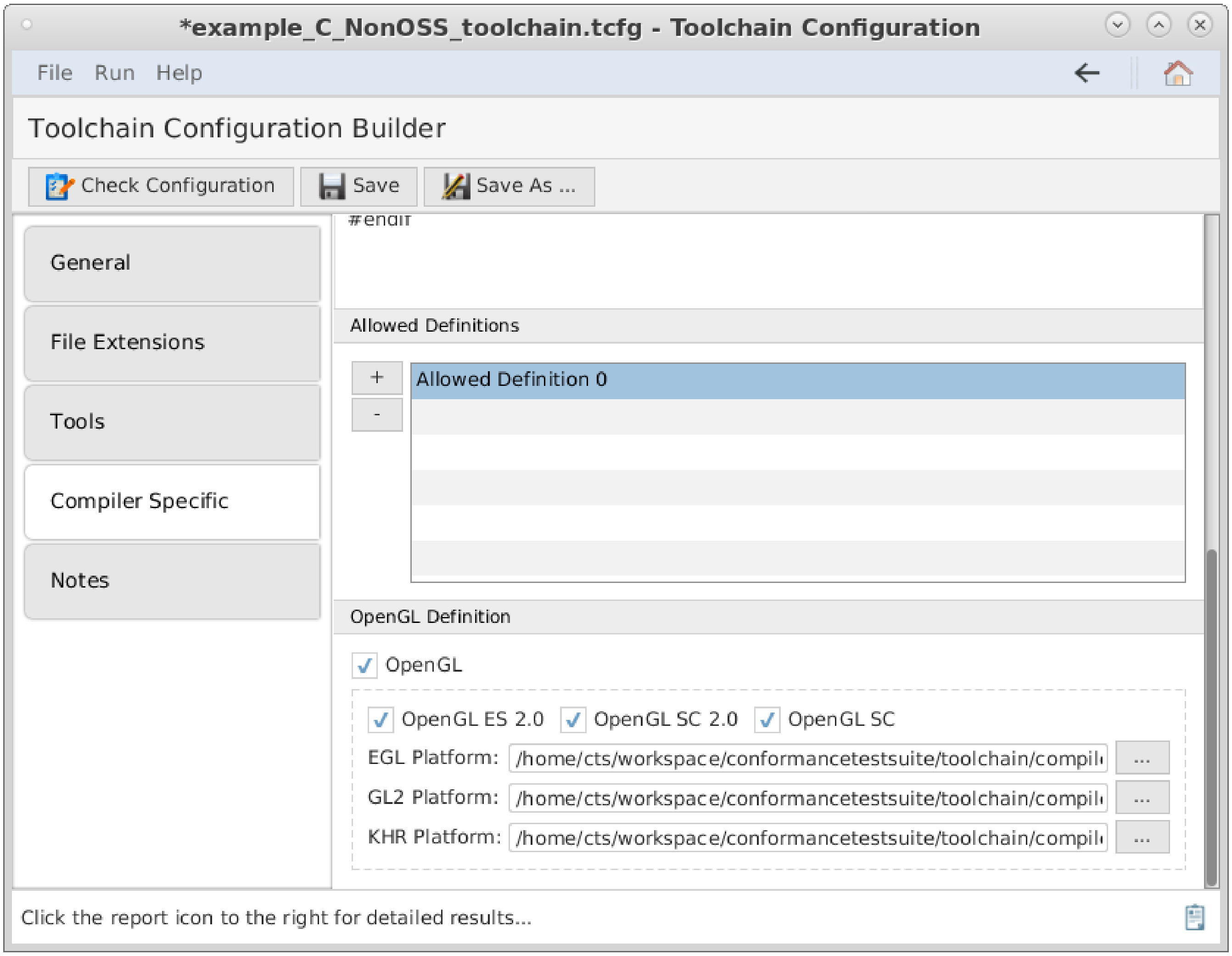
```
#ifdef __cplusplus
        // Define NULL for C++ as the constant 0 with no specific type.
        #define FACE_NULL 0
#else
        // Define NULL for C as 0 cast to a void pointer.
        #define FACE_NULL ( (void*) 0 )
#endif
```

Allowed Definition 0

☑ OpenGL

Click the report icon to the right for detailed results...

18. Add allowed definitions by scrolling down to the Allowed Definitions section shown below.



#endif

Allowed Definition 0

☑ OpenGL

☑ OpenGL ES 2.0 ☑ OpenGL SC 2.0 ☑ OpenGL SC

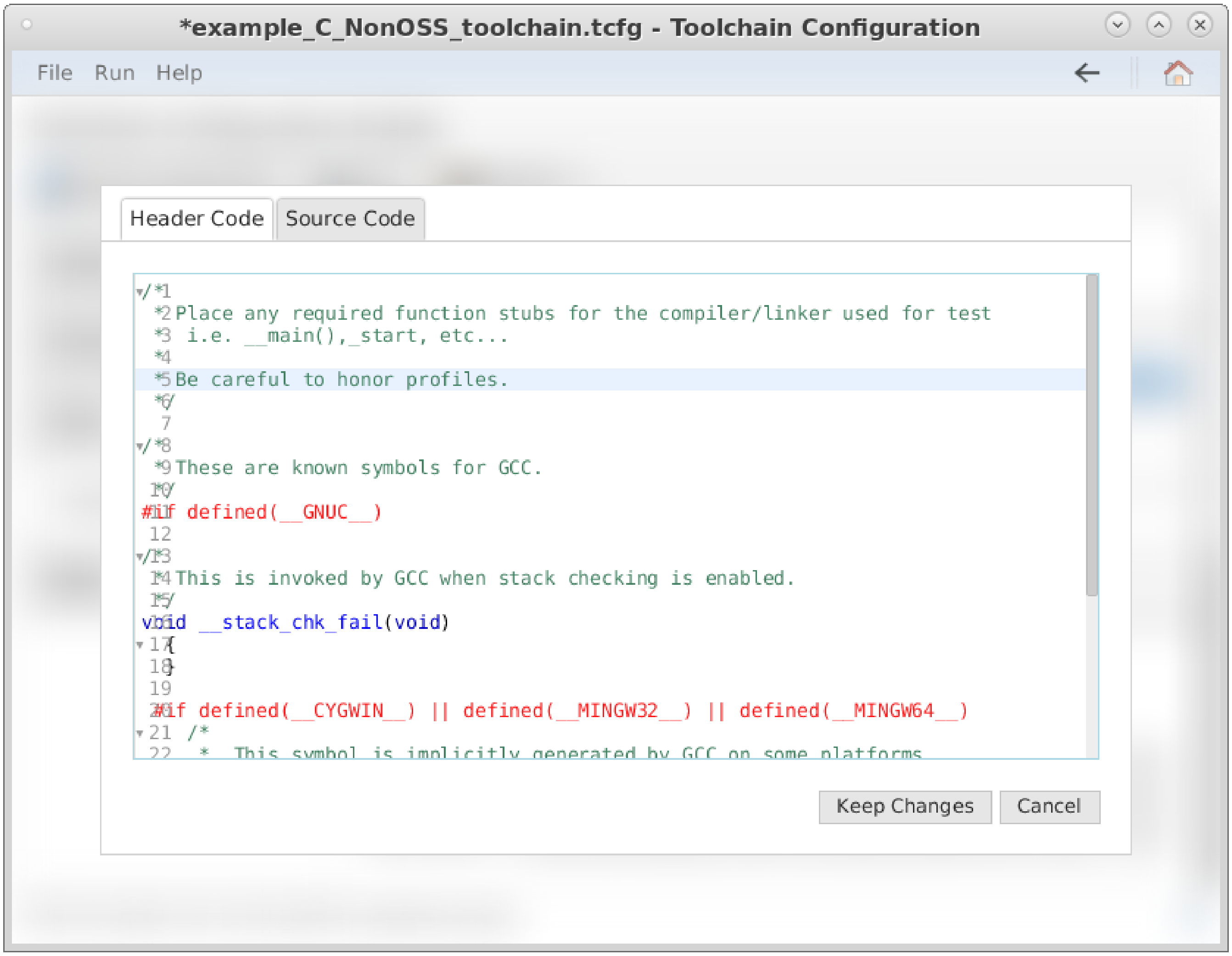
EGL Platform: /home/cts/workspace/conformancetestsuite/toolchain/compil

GL2 Platform: /home/cts/workspace/conformancetestsuite/toolchain/compil

KHR Platform: /home/cts/workspace/conformancetestsuite/toolchain/compil

Click the report icon to the right for detailed results...

Click the + . Then double click on the definition created in the list box to the right. You will then see the screen shown below (example code from a sample toolchain is shown).

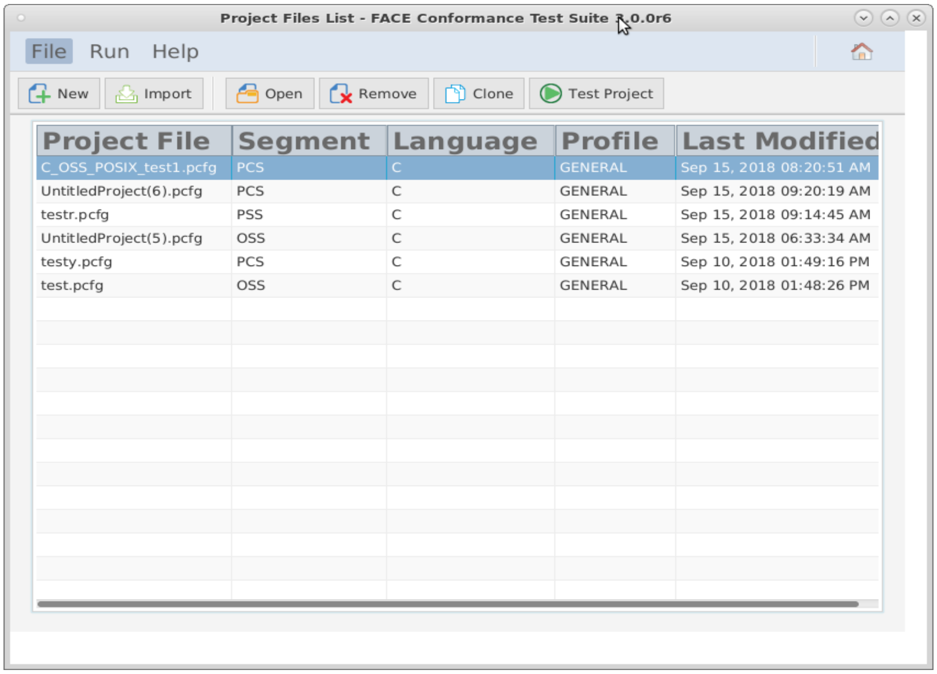


```
/*
 * Place any required function stubs for the compiler/linker used for test
 * i.e. __main(), _start, etc...
 *
 * Be careful to honor profiles.
 */

/*
 * These are known symbols for GCC.
 */
#if defined(__GNUC__)

/*
 * This is invoked by GCC when stack checking is enabled.
 */
void __stack_chk_fail(void)
{
}

#if defined(__CYGWIN__) || defined(__MINGW32__) || defined(__MINGW64__)
/*
 *  This symbol is implicitly generated by GCC on some platforms.
```

Add the code to the header and source tabs by editing the text area. Refer to *Compiler Specific Functionality* section.

Project Files List

The Project Files List can be accessed by selecting the File→Projects option from the top menu option.



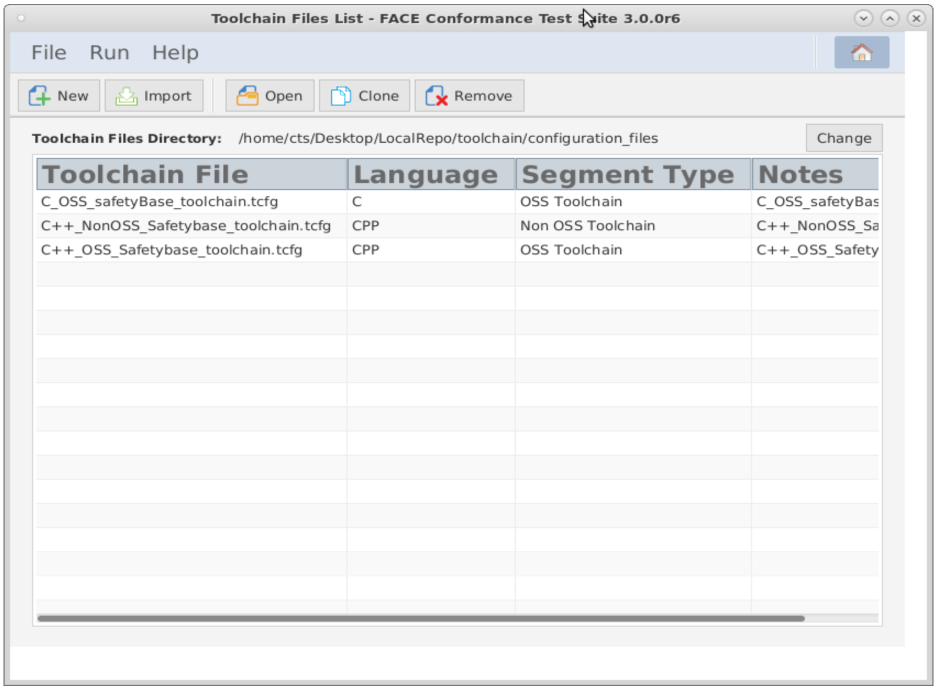
Project File	Segment	Language	Profile	Last Modified
C_OSS_POSIX_test1.pcfg	PCS	C	GENERAL	Sep 15, 2018 08:20:51 AM
UntitledProject(6).pcfg	PCS	C	GENERAL	Sep 15, 2018 09:20:19 AM
testr.pcfg	PSS	C	GENERAL	Sep 15, 2018 09:14:45 AM
UntitledProject(5).pcfg	OSS	C	GENERAL	Sep 15, 2018 06:33:34 AM
testy.pcfg	PCS	C	GENERAL	Sep 10, 2018 01:49:16 PM
test.pcfg	OSS	C	GENERAL	Sep 10, 2018 01:48:26 PM

This view represents a working sandbox view of project files that have been created or recently edited. Selecting a project from this list view will provide access to the following options:

- **New** – Opens a new file dialog to create an empty project from scratch.
- **Import** – Provides a file browser dialog to allow the user to find an existing project file and allow it to be added to the list view.
- **Open –** Opens the currently selected project from the list view.
- **Remove** – Removes the currently selected project from the list view. (Note that this option does not delete the project but removes it from the list view only)
- **Clone** – Creates a copy of the currently selected project and saves it at the same location as the original
- **Test Project** – Executes the test procedure on the currently selected project.

Toolchain Files List

The Toolchain Files List can be accessed by selecting the File → Toolchains option from the top menu option.



Toolchain Files Directory: /home/cts/Desktop/LocalRepo/toolchain/configuration_files

Toolchain File	Language	Segment Type	Notes
C_OSS_safetyBase_toolchain.tcfg	C	OSS Toolchain	C_OSS_safetyBas
C++_NonOSS_Safetybase_toolchain.tcfg	CPP	Non OSS Toolchain	C++_NonOSS_Sa
C++_OSS_Safetybase_toolchain.tcfg	CPP	OSS Toolchain	C++_OSS_Safety

This view represents a working sandbox view of toolchain files that have been created or recently edited within the "Toolchain Files Directory". The following options are available either by default or when you select a toolchain from the list view:

- **New** – Opens the toolchain editor for creation of a new toolchain from scratch.
- **Import** – Provides a file browser dialog to allow the user to find and select 1 or more existing toolchain file(s) and allow it to be copied to the "Toolchain Files Directory" displayed above the list view. (Note that this option is different from the Project Files List's Import function because you are making a copy of an existing toolchain from another location to the working directory location. If you intend on modifying a toolchain that isn't located in the working toolchain directory, then it is best to change the working toolchain directory to be that of the directory from which the toolchain resides in.)
- **Open –** Opens the currently selected toolchain from the list view into the toolchain editor.
- **Remove** – Removes the currently selected toolchain from the list view. (Note that this option does not delete the toolchain but removes it from the list view only.)
- **Clone** – Creates a copy of the currently selected toolchain and saves it to the "Toolchain Files Directory".
- **Change** – This opens a directory browser dialog to allow the user to change the directory to search for and display all available toolchains in this toolchain list view.

Compiler Specific Functionality

There is compiler specific information that will be needed to conduct conformance tests. This information is stored in the compilerSpecific subdirectory. In particular, the mapping between standard types in C/C++ and their exact definitions for a given compiler will need to be specified. There may also be compiler specific built-in functions/methods that cause linker errors even when compiler and linking against the OS gold standard libraries (i.e. __main, __stack_chk_fail). There may be valid graphics related calls that are not called out specifically in the standard.

You may add allowed functions to the conformance test by editing the “CompilerSpecific” source file. This file can be edited in the Compiler Specific section of the toolchain configuration builder. See *Installation and Configuration* section above for details. You only need to add a function stub, since these conformance test objects are never actually executed. The compiler specific source file is included in the conformance report to show any functions that were used.

To specify the language mappings, the exact types need to be added to the toolchain file. This can be done through the toolchain configuration. See the *Installation and Configuration* section above for details.

Compiler specific methods must be reported to the Verification Authority (VA) (and are included in the Test Suite results).

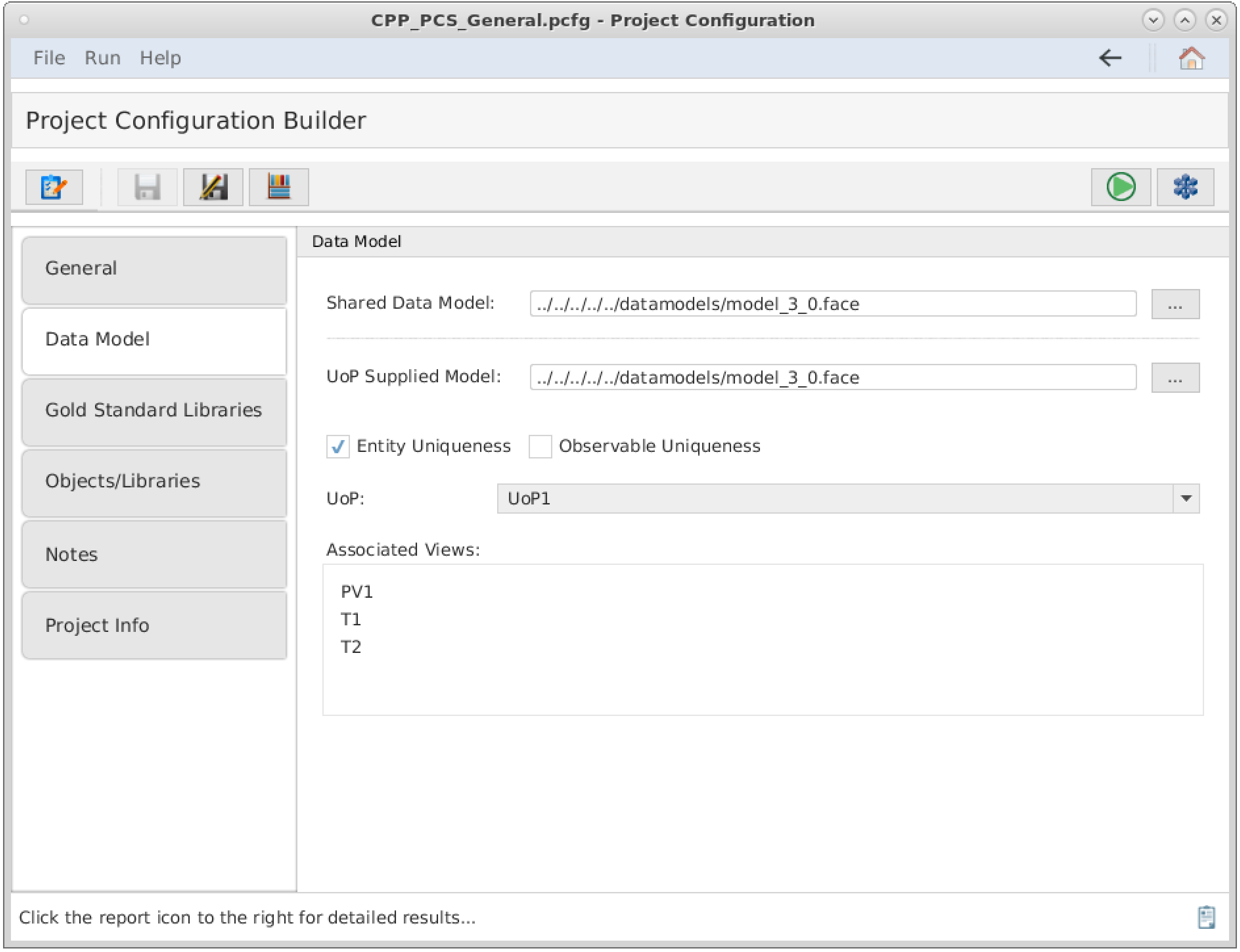
Testing a Portable Components Segment (PCS) Application

What You Must Provide

- Your project's object files (C/C++/Ada) or class files (Java). Alternatively, source files can be provided and the CTS will build them into object or class files using the toolchain configuration.
- Your project's header files (C/C++) or spec files (Ada).
- Your project's data model.
- Your project's toolchain file.

Procedure

1. Complete the procedure in the *Initializing the Conformance Test Suite* section above.
2. Select the Data Model tab to display the data model information below.



Data Model

Shared Data Model: ../../../../../datamodels/model_3_0.face

UoP Supplied Model: ../../../../../datamodels/model_3_0.face

☑ Entity Uniqueness ☐ Observable Uniqueness

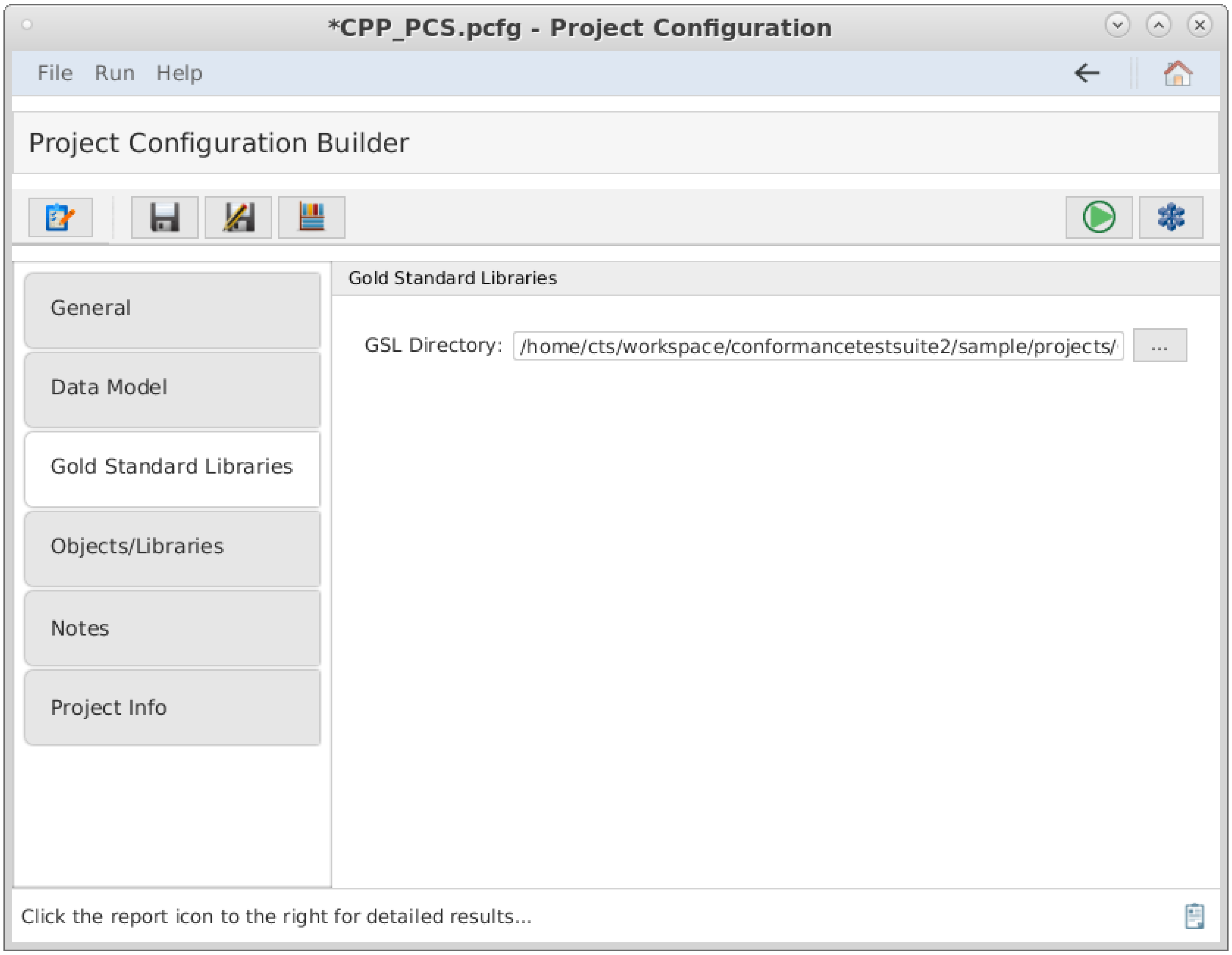
UoP: UoP1

Associated Views:

PV1
T1
T2

Click the report icon to the right for detailed results...

3. Set the path to the Shared Data Model and UoP Supplied Model. **Note: This directory is relative to the base directory set in the General tab.**
4. Select the Gold Standard Libraries tab to display the options below.

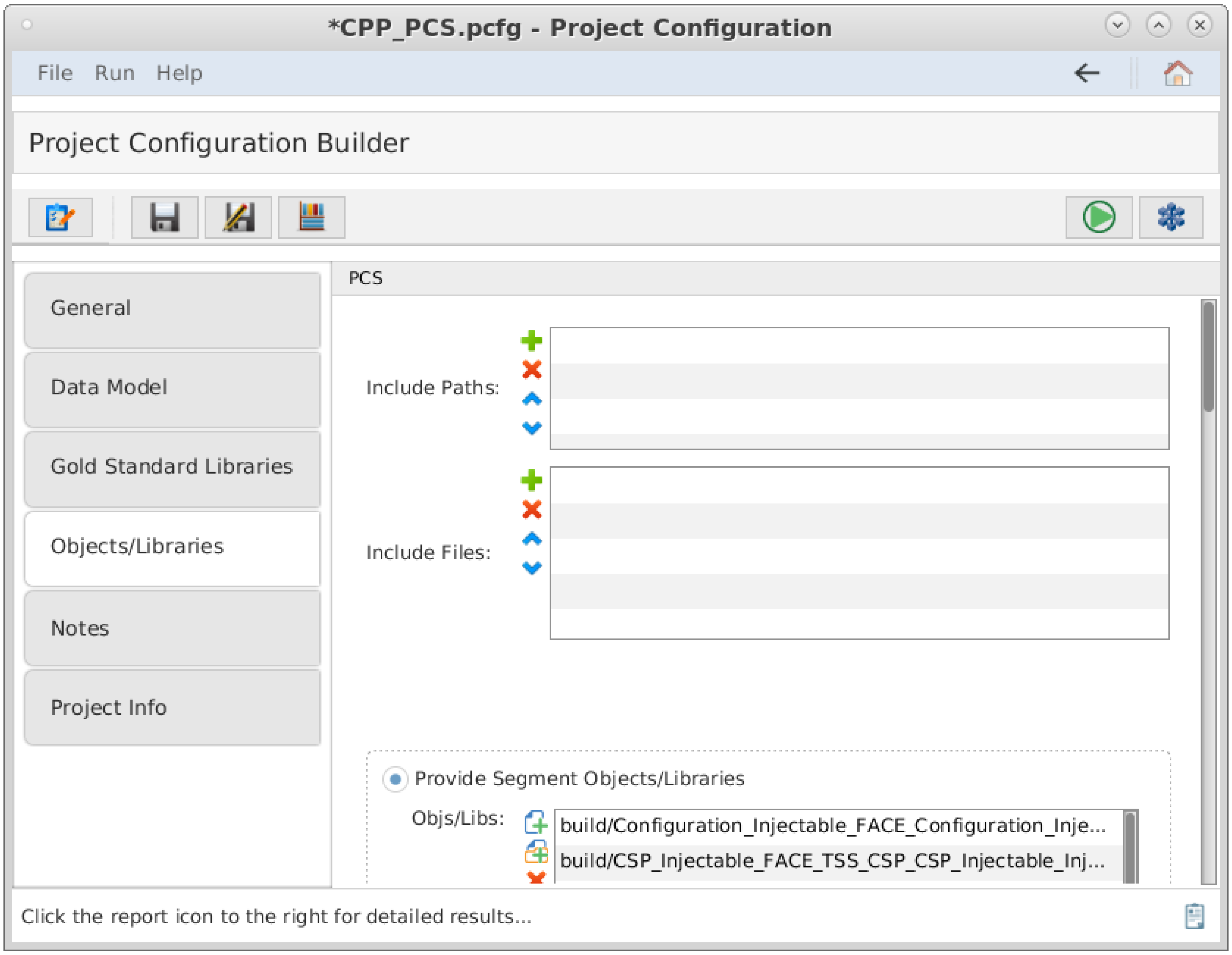


GSL Directory: /home/cts/workspace/conformancetestsuite2/sample/projects/

Click the report icon to the right for detailed results...

5. Set the directory where the gold standard libraries will be generated and stored for the test. **Note: This directory is relative to the base directory set in the General tab.**

6. Select the Objects/Libraries tab to display the portable components information options shown below. For C/C++ projects, use the list boxes to add the include files and include paths (if applicable) for the UoC for the concrete interface implementations provided by the UoC. All 'include files' must exist in one of the 'include paths' specified here. Any files included by the Factory Function Source File (see next few steps in this tutorial) must exist in one of the Include Paths specified.



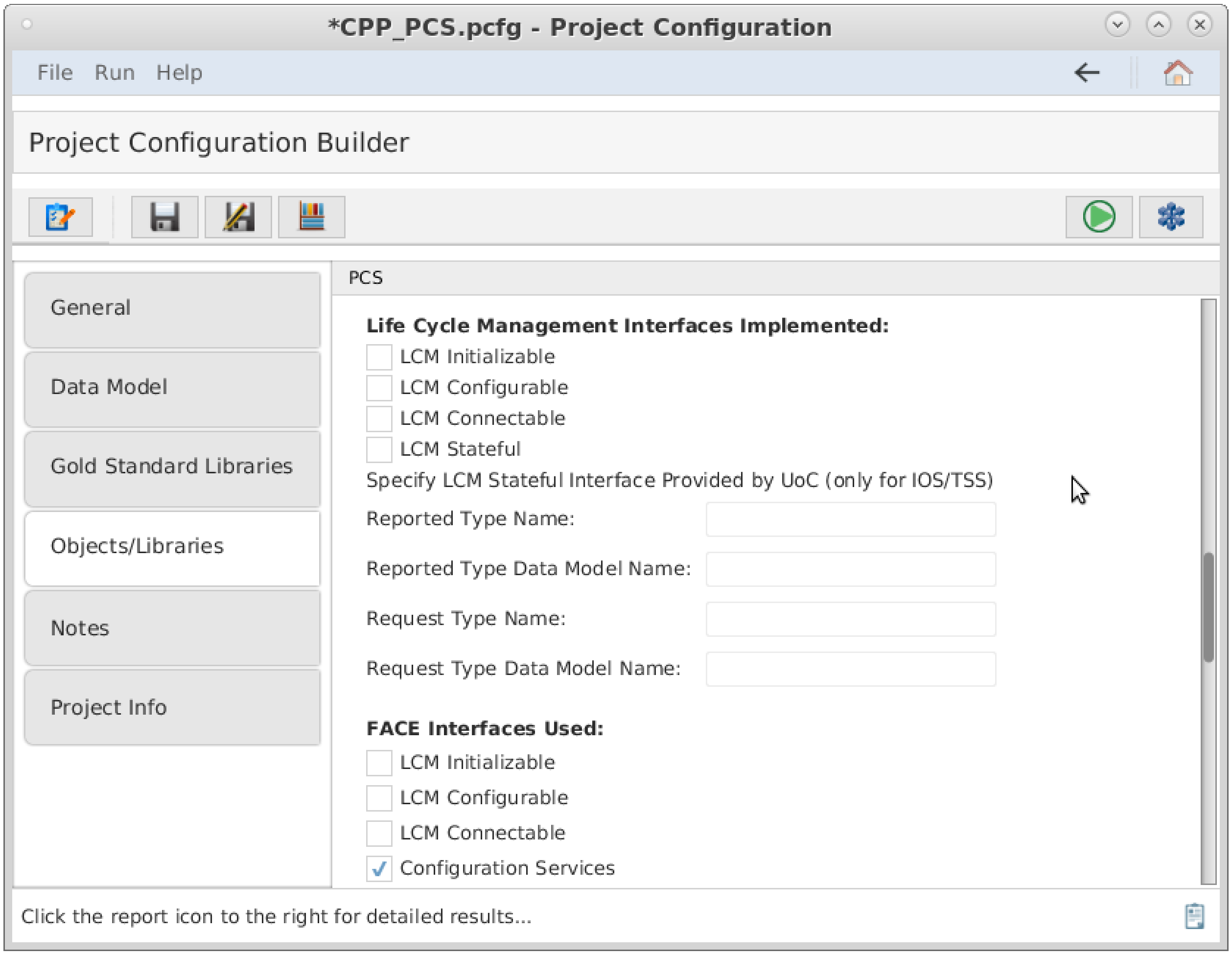
● Provide Segment Objects/Libraries

Objs/Libs:
build/Configuration_Injectable_FACE_Configuration_Inje...
build/CSP_Injectable_FACE_TSS_CSP_CSP_Injectable_Inj...

Click the report icon to the right for detailed results...

7. If providing object files for your UoC: before providing your object or library files, you must build them against the Gold Standard Library headers. The CTS will generate the FACE headers for any interface your UoC uses so that you can build your source code against those. Skip selecting your UoC's object files until you have generated the GSLs and FACE headers and have built your UoC code against these headers. Therefore, skip the section on selecting object files for now.

8. Scroll down and select any of the FACE interfaces the UoC implements. If the Stateful interface is implemented and the project is a IOSS or TSS project, enter the datamodel name and datatype name of the reported and request datatype. For PSSS and PCS projects, the datatypes are defined by the architecture model selected.



PCS

Life Cycle Management Interfaces Implemented:

☐ LCM Initializable
☐ LCM Configurable
☐ LCM Connectable
☐ LCM Stateful

Specify LCM Stateful Interface Provided by UoC (only for IOS/TSS)

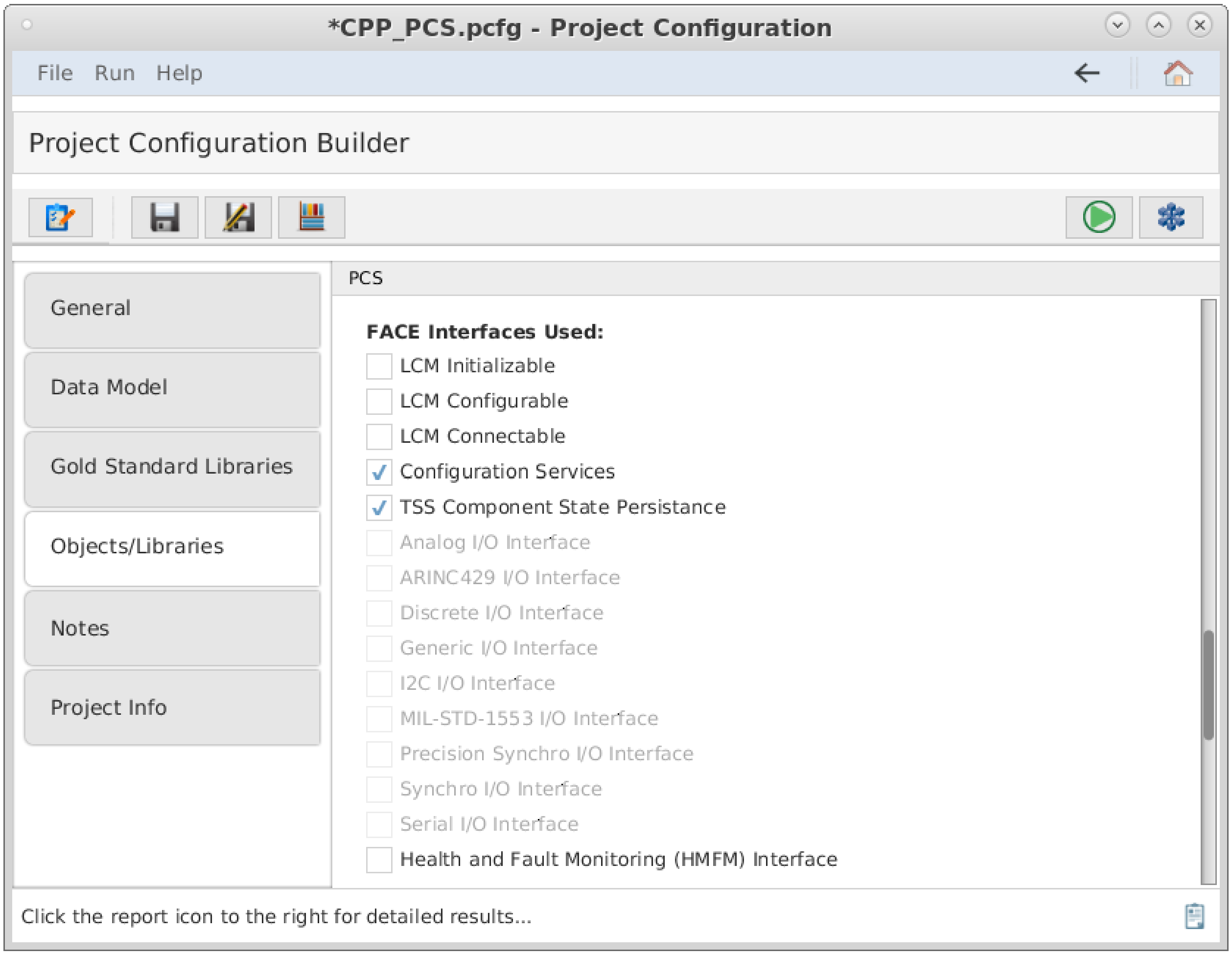
Reported Type Name:
Reported Type Data Model Name:
Request Type Name:
Request Type Data Model Name:

FACE Interfaces Used:

☐ LCM Initializable
☐ LCM Configurable
☐ LCM Connectable
☑ Configuration Services

Click the report icon to the right for detailed results...

9. Scroll down and select the FACE interfaces that are used by the UoC. The UoC must provide an Injectable interface for each FACE interface it uses. By specifying that the UoC under test "uses" a given interface, this indicates that it implements an Injectable interface for that interface and this will be tested by the CTS.

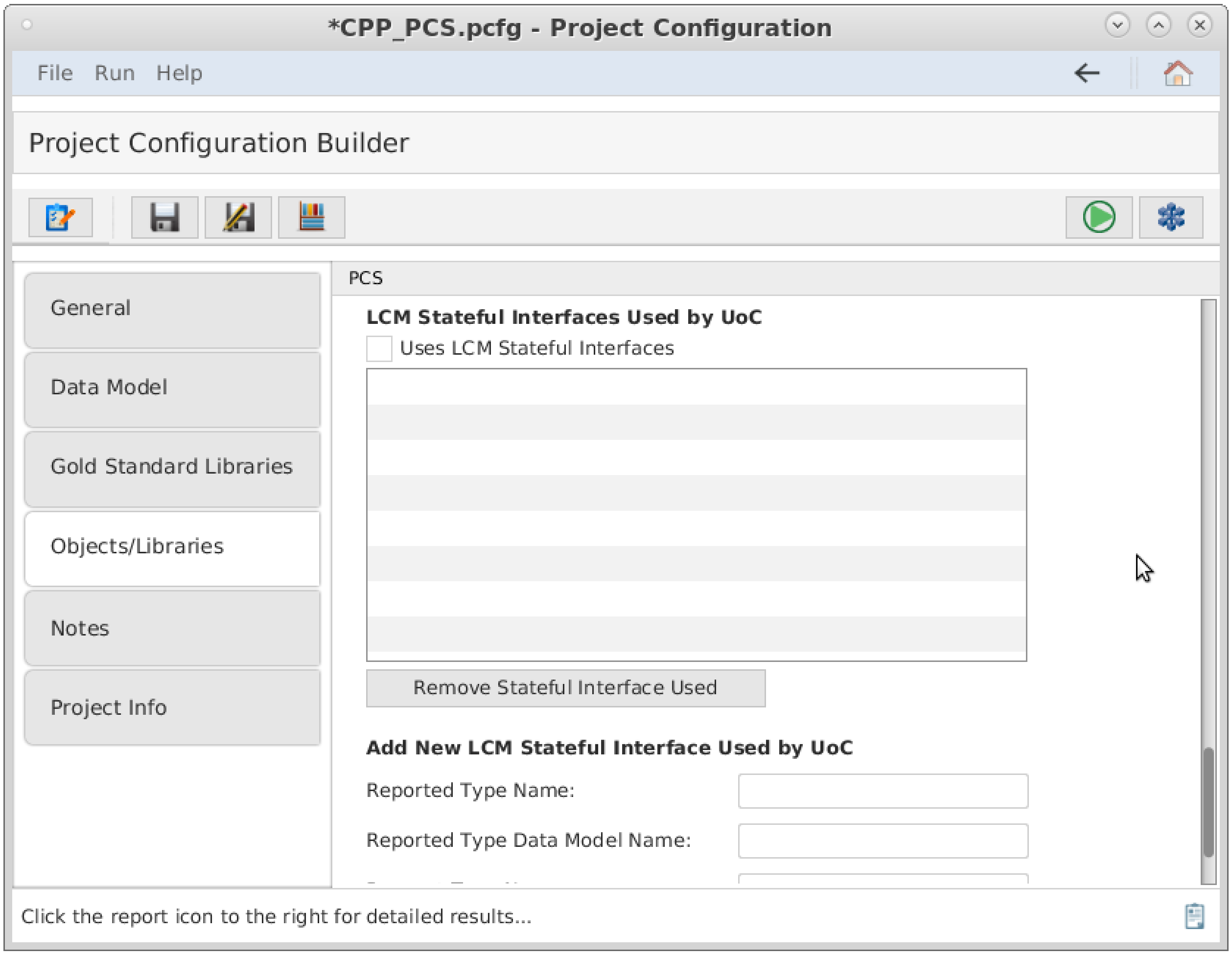


FACE Interfaces Used:

☐ LCM Initializable
☐ LCM Configurable
☐ LCM Connectable
☑ Configuration Services
☑ TSS Component State Persistance
☐ Analog I/O Interface
☐ ARINC429 I/O Interface
☐ Discrete I/O Interface
☐ Generic I/O Interface
☐ I2C I/O Interface
☐ MIL-STD-1553 I/O Interface
☐ Precision Synchro I/O Interface
☐ Synchro I/O Interface
☐ Serial I/O Interface
☐ Health and Fault Monitoring (HMFM) Interface

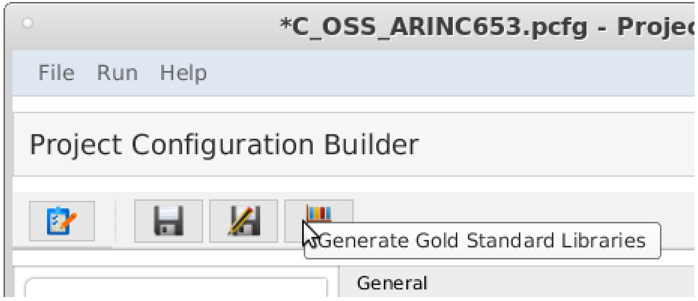
Click the report icon to the right for detailed results...

10. Scroll down and enter the information regarding any FACE Life Cycle Management Stateful interfaces that are used by the UoC. If none are used, leave this section blank. Multiple Life Cycle Management Stateful interfaces may be used by a UoC. The required information for each Stateful interface is: the data model and datatype name of the reported datatype, and the data model and datatype name of the request datatype.


*CPP_PCS.pcfg - Project Configuration
File
Run
Help
Project Configuration Builder
General
Data Model
Gold Standard Libraries
Objects/Libraries
Notes
Project Info
PCS
LCM Stateful Interfaces Used by UoC
☐ Uses LCM Stateful Interfaces
Remove Stateful Interface Used
Add New LCM Stateful Interface Used by UoC
Reported Type Name:
Reported Type Data Model Name:

Click the report icon to the right for detailed results...

11. In order to build your source code (assuming you are providing the test object code), you will need FACE interface headers for any interfaces your UoC uses. For example, for a PCS you will need any TSS headers your code uses, as standardized in the FACE standard. The CTS will generate these headers for you. Click the Generate GSLs Button in the upper left corner of the window as shown below to generate the FACE headers as well as the gold standard libraries (GSLs).

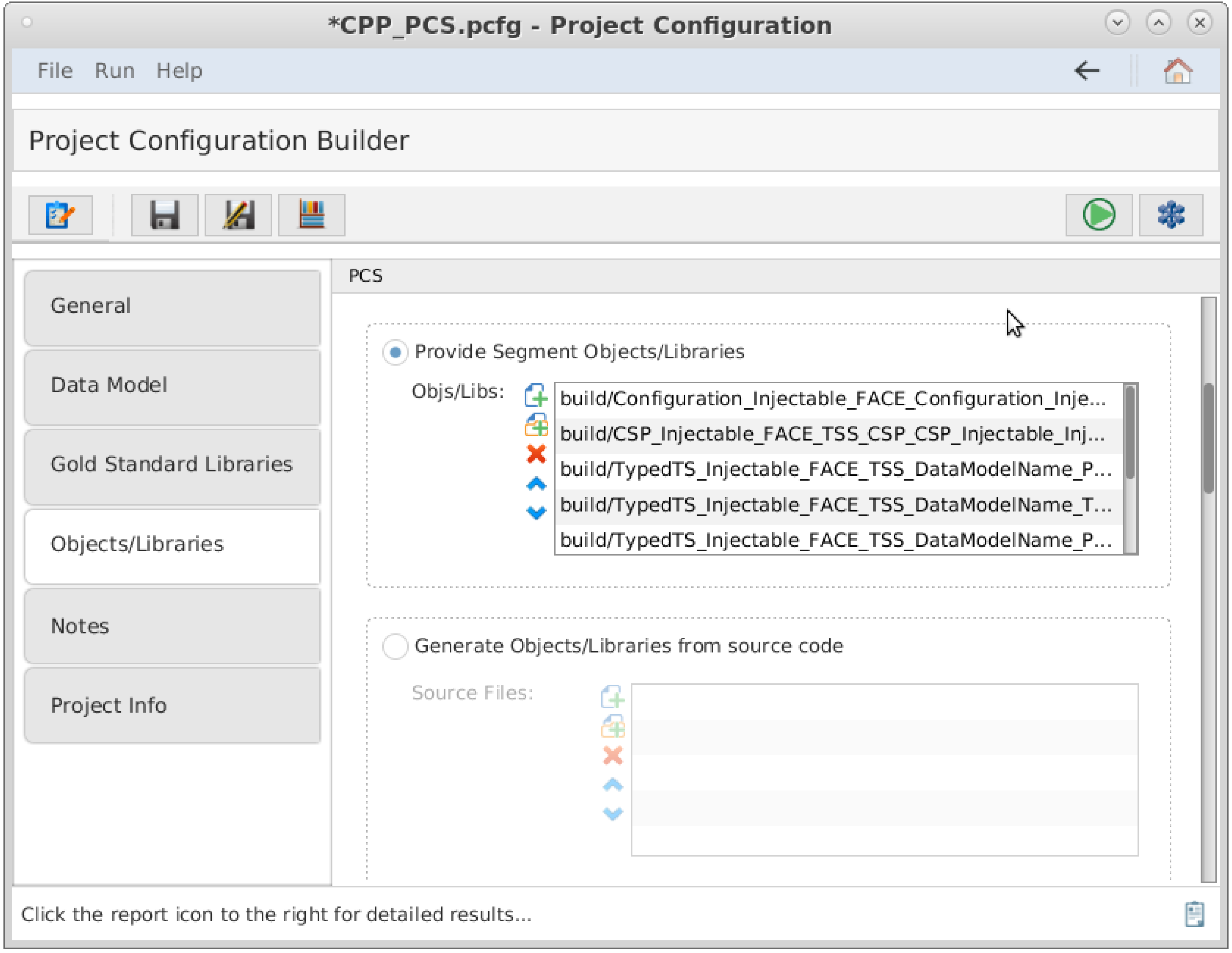

*C_OSS_ARINC653.pcfg - Projec
File
Run
Help
Project Configuration Builder
Generate Gold Standard Libraries
General

12. If providing objects for your UoC, you may now build your objects using the FACE headers generated into the GSL directory's include/FACE subdirectory. Examine the contents of this directory to see the standardized header names of all non-OSS FACE interfaces. Note that for C/C++ all FACE headers should be included with the relative path starting from the FACE directory, for example: "FACE/IOSS/Analog.hpp" The include directories to be used are described in a generated README file in the GSL directory. OSS include directories are listed and are in specific subfolders of the CTS folder goldStandardLibraries.

    Therefore, when building your object code for your UoC, you will need to include these subfolders and include any FACE headers as "FACE/[name of header file]". Note that the headers in this subdirectory are deleted each time the test is run, so do not store any project files in this directory.

    The GSL libraries will be generated into the GSL Directory. You may wish to use the GSLs during development to check that your code builds against them, but there is no need to include them in your CTS project. The CTS will rebuild the appropriate GSLs when you run the test and link them as part of the test.

13. Return to the list of segment objects/libraries or source lists as shown below.



● Provide Segment Objects/Libraries

Objs/Libs:
build/Configuration_Injectable_FACE_Configuration_Inje...
build/CSP_Injectable_FACE_TSS_CSP_CSP_Injectable_Inj...
build/TypedTS_Injectable_FACE_TSS_DataModelName_P...
build/TypedTS_Injectable_FACE_TSS_DataModelName_T...
build/TypedTS_Injectable_FACE_TSS_DataModelName_P...

○ Generate Objects/Libraries from source code

Source Files:

Click the report icon to the right for detailed results...

14. Enter the full pathnames of your project's object and/or library files. You may add each object file. You may also choose the directory where object files are located. All object files in the directory specified as well as object files in subdirectories will be chosen. Currently, library files must be chosen individually, not by directory. A combination of directories and object/library files may be specified.

15. **Important (applies to all languages):** You must provide a source file that contains the implementation for each of the factory functions that creates an instance of each interface your UoC is providing. To determine which factory functions are necessary, complete the rest of this tutorial and click the "Generate GSLs" button (library icon) in the toolbar. Then, find the generated header/spec file in the generated subfolder 'include/FACE' within the Gold Standard Libraries folder you specified for the project. For C/C++, this file (CTS_Factory_Functions .h or .hpp) will contain the declarations of all expected factory functions that the CTS requires for testing your UoC. You must provide a source file that implements each of these functions. The implementation for each function must instantiate the corresponding concrete class and return a pointer to that object (the return type will be a pointer to the FACE base class). Returning a null pointer is not acceptable. This source file must be provided with your project and will be reviewed to ensure it instantiates your UoC's concrete class for that interface. For Ada, this will generate a spec file (.ads) cts_factory_functions.ads which has the procedures you must implement in the source file. The source file for Ada must be named "cts_factory_functions.adb" and implement each of these procedures, returning a concrete version of each type as an access type. In the text field labeled "Source file with factory functions for interfaces provided (C/C++/Ada only)", use the '...' button to the right to browse for the source file. **Any header files included by the source file must exist in one of the Include Paths specified above.** For Java, a source file named CTS_Factory_Functions.java will be generated in the factory/ subfolder (package subfolder) You must fill in the implementation of each function, add any imports, and add this to your project in the CTS in this file field. Note that this file will NOT be overwritten so if the interfaces the UoC uses have changed, you must delete your file in order to regenerate a new one with the new set of functions to implement.

16. Select 📝 to verify that the Project Configuration File is valid.

17. Click the ▶ button at the top of the screen to test the segment. (This may take a few minutes). The results will be written to a

PDF file. The directory of the results file will be located in the directory of the project configuration file. This directory path will be listed in the "Output File Location" of the Conformance Test Results page.

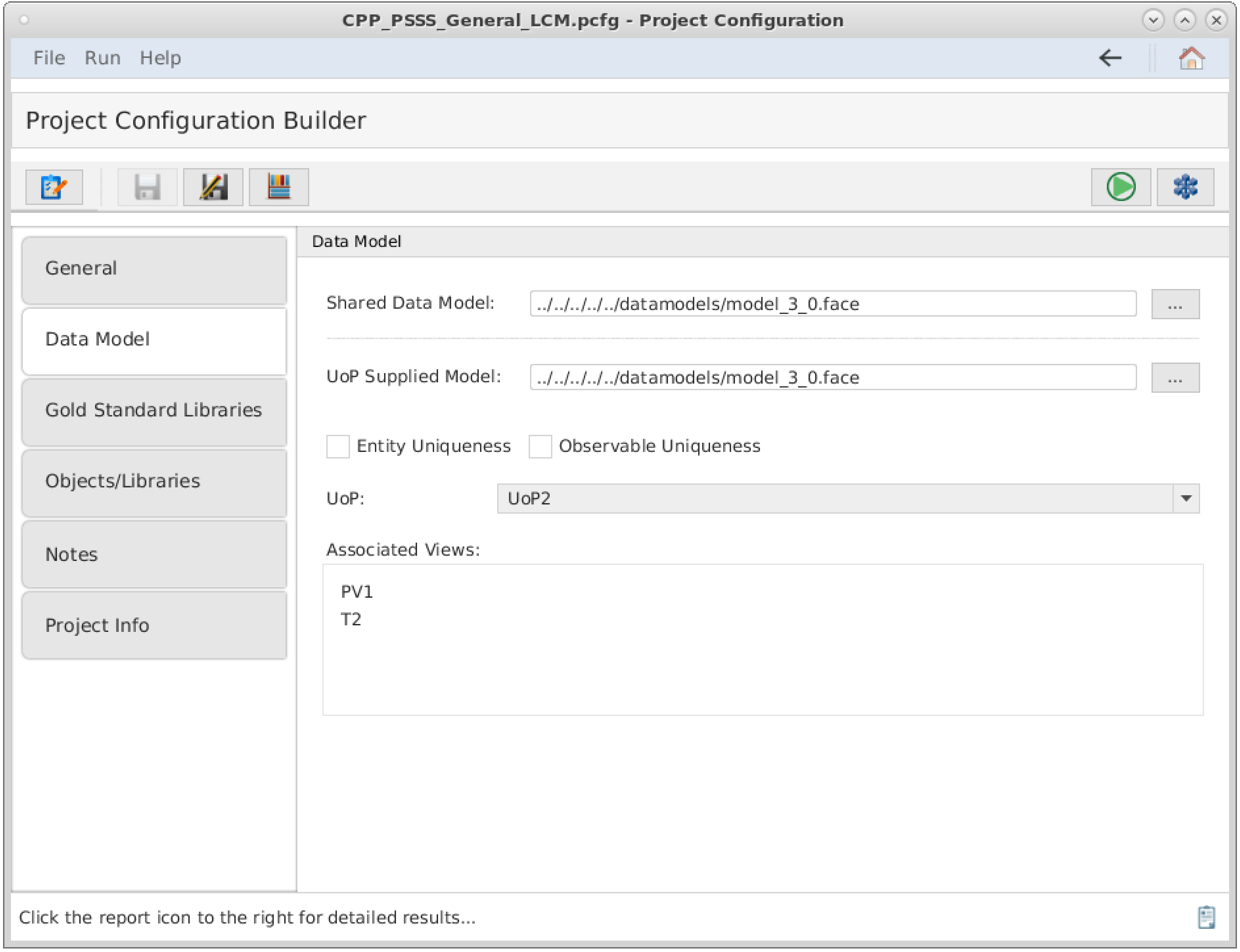
Testing a Platform Specific Services Segment (PSSS) Application

What You Must Provide

- Your project's object files (C/C++/Ada) or class files (Java). Alternatively, source files can be provided and the CTS will build them into object or class files using the toolchain configuration.
- Your project's header files (C/C++) or spec files (Ada).
- Your project's data model.
- Your project's toolchain file.

Procedure

1. Complete the procedure in the *Initializing the Conformance Test Suite* section above.
2. Select the Data Model tab to display the data model information below.



Data Model

Shared Data Model: ../../../../../datamodels/model_3_0.face

UoP Supplied Model: ../../../../../datamodels/model_3_0.face

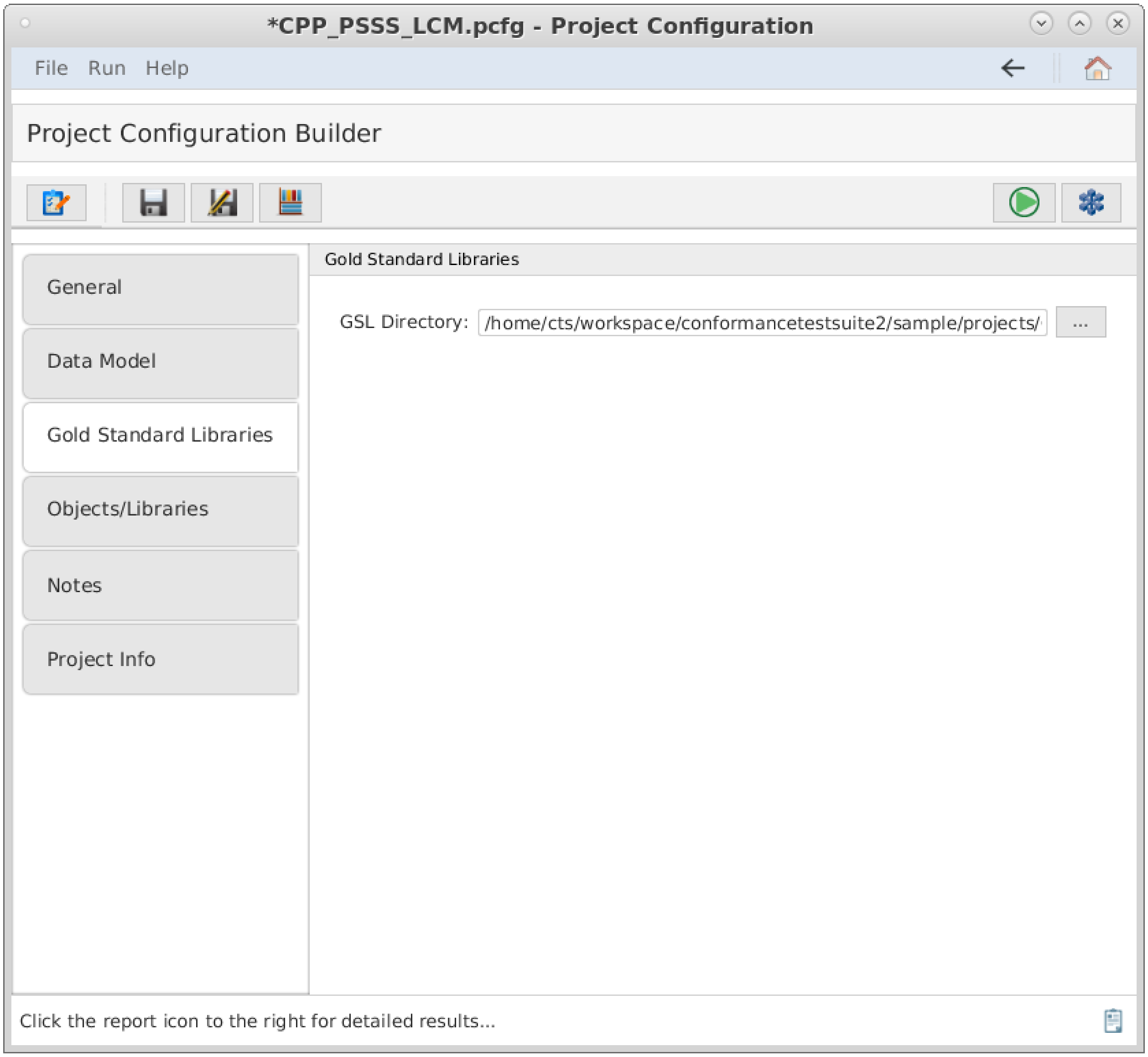
☐ Entity Uniqueness ☐ Observable Uniqueness

UoP: UoP2

Associated Views:
PV1
T2

Click the report icon to the right for detailed results...

3. Set the path to the Shared Data Model and UoP Supplied Model. **Note: This directory is relative to the base directory set in the General tab.**
4. Select the Gold Standard Libraries tab to display the options below.

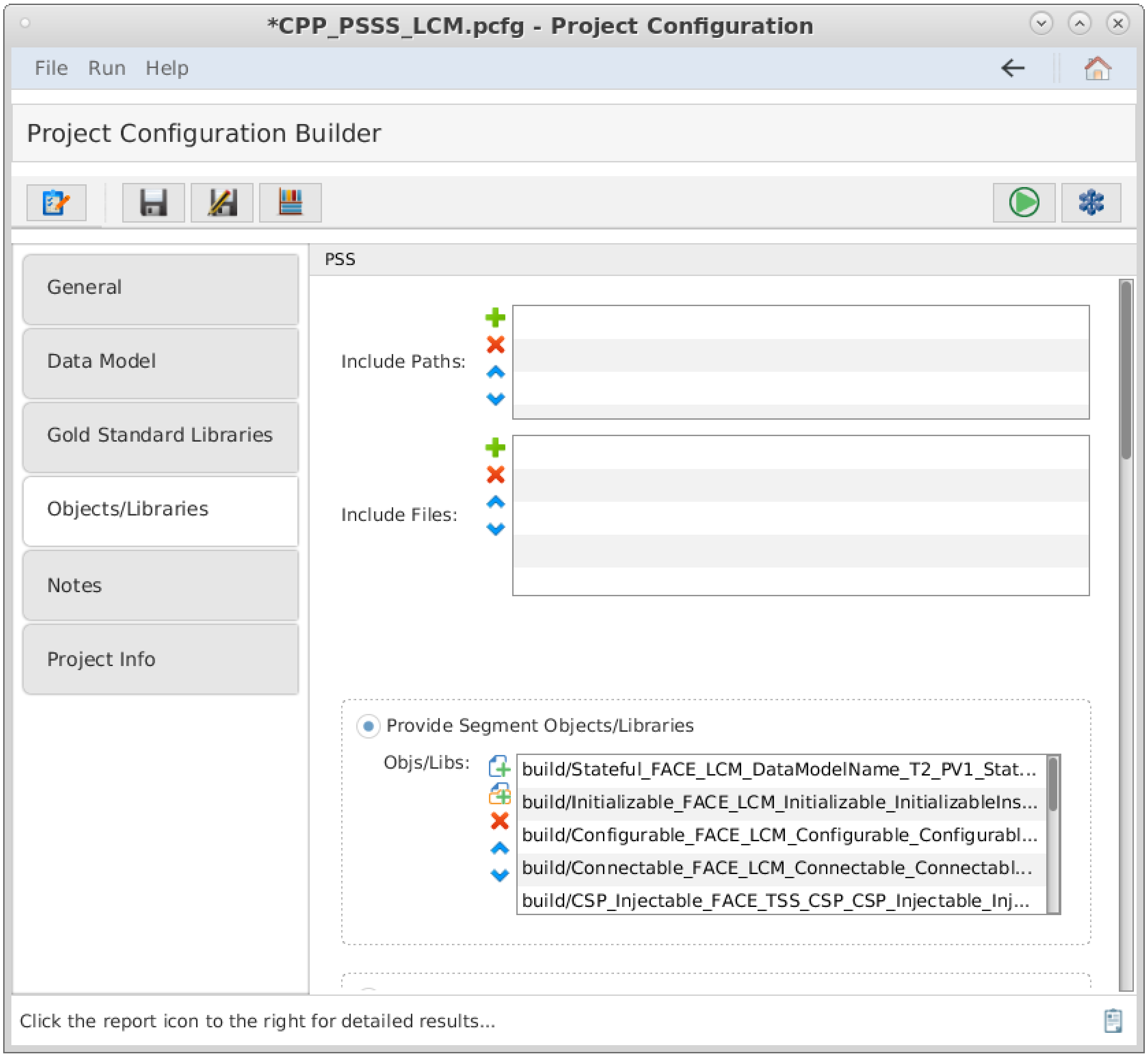


GSL Directory: /home/cts/workspace/conformancetestsuite2/sample/projects/

Click the report icon to the right for detailed results...

5. Set the directory where the gold standard libraries and the FACE interface headers your code uses will be generated and stored. **Note: This directory is relative to the base directory set in the General tab.**

6. Select the Objects/Libraries tab to display the portable components information options shown below. For C/C++ projects, use the list boxes to add the include files and include paths (if applicable) for the UoC for the concrete interface implementations provided by the UoC. All 'include files' must exist in one of the 'include paths' specified here. Any files included by the Factory Function Source File (see next few steps in this tutorial) must exist in one of the Include Paths specified.



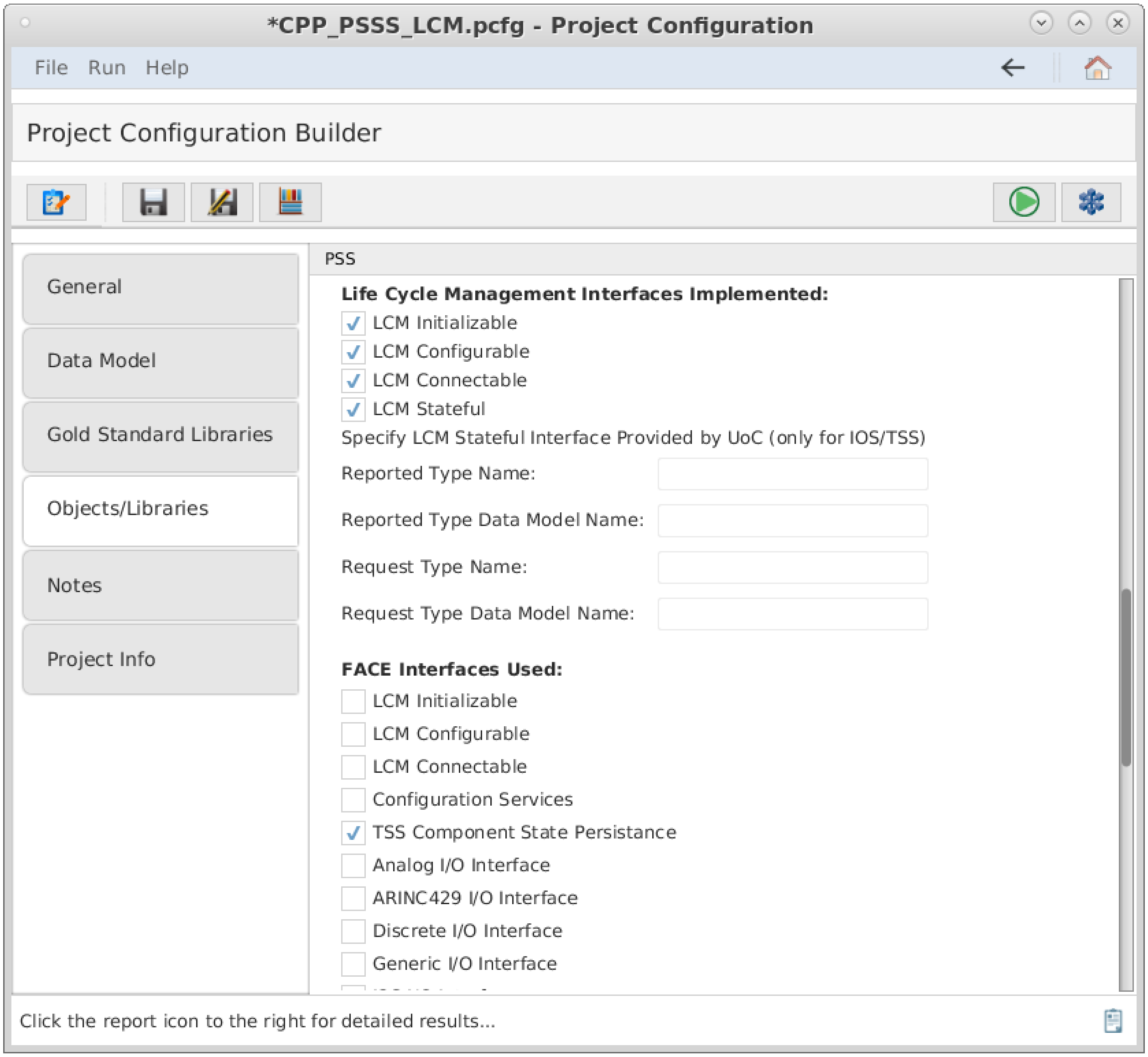
● Provide Segment Objects/Libraries

Objs/Libs:
build/Stateful_FACE_LCM_DataModelName_T2_PV1_Stat...
build/Initializable_FACE_LCM_Initializable_InitializableIns...
build/Configurable_FACE_LCM_Configurable_Configurabl...
build/Connectable_FACE_LCM_Connectable_Connectabl...
build/CSP_Injectable_FACE_TSS_CSP_CSP_Injectable_Inj...

Click the report icon to the right for detailed results...

7. If providing object files for your UoC: before providing your object or library files, you must build them against the Gold Standard Library headers. The CTS will generate the FACE headers for any interface your UoC uses so that you can build your source code against those. Skip selecting your UoC's object files until you have generated the GSLs and FACE headers and have built your UoC code against these headers. Therefore, skip the section on selecting object files for now.

8. Scroll down and select any of the FACE interfaces the UoC implements. If the Stateful interface is implemented and the project is a IOSS or TSS project, enter the datamodel name and datatype name of the reported and request datatype. For PSSS and PCS projects, the datatypes are defined by the architecture model selected.



Life Cycle Management Interfaces Implemented:

☑ LCM Initializable
☑ LCM Configurable
☑ LCM Connectable
☑ LCM Stateful

Specify LCM Stateful Interface Provided by UoC (only for IOS/TSS)

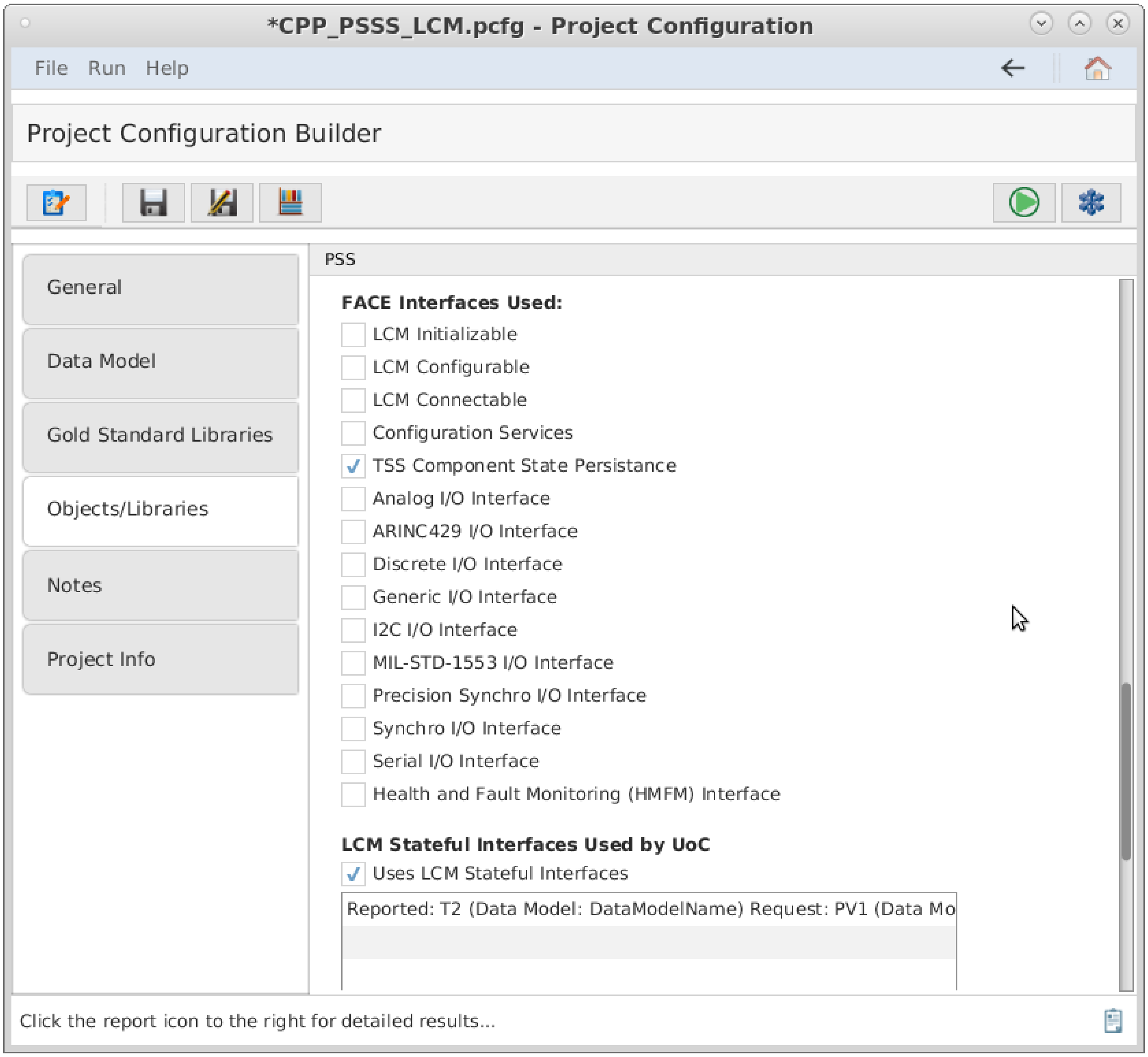
Reported Type Name:
Reported Type Data Model Name:
Request Type Name:
Request Type Data Model Name:

FACE Interfaces Used:

☐ LCM Initializable
☐ LCM Configurable
☐ LCM Connectable
☐ Configuration Services
☑ TSS Component State Persistance
☐ Analog I/O Interface
☐ ARINC429 I/O Interface
☐ Discrete I/O Interface
☐ Generic I/O Interface

Click the report icon to the right for detailed results...

9. Scroll down and select the FACE interfaces that are used by the UoC. The UoC must provide an Injectable interface for each FACE interface it uses. By specifying that the UoC under test "uses" a given interface, this indicates that it implements an Injectable interface for that interface and this will be tested by the CTS.



FACE Interfaces Used:

☐ LCM Initializable
☐ LCM Configurable
☐ LCM Connectable
☐ Configuration Services
☑ TSS Component State Persistance
☐ Analog I/O Interface
☐ ARINC429 I/O Interface
☐ Discrete I/O Interface
☐ Generic I/O Interface
☐ I2C I/O Interface
☐ MIL-STD-1553 I/O Interface
☐ Precision Synchro I/O Interface
☐ Synchro I/O Interface
☐ Serial I/O Interface
☐ Health and Fault Monitoring (HMFM) Interface

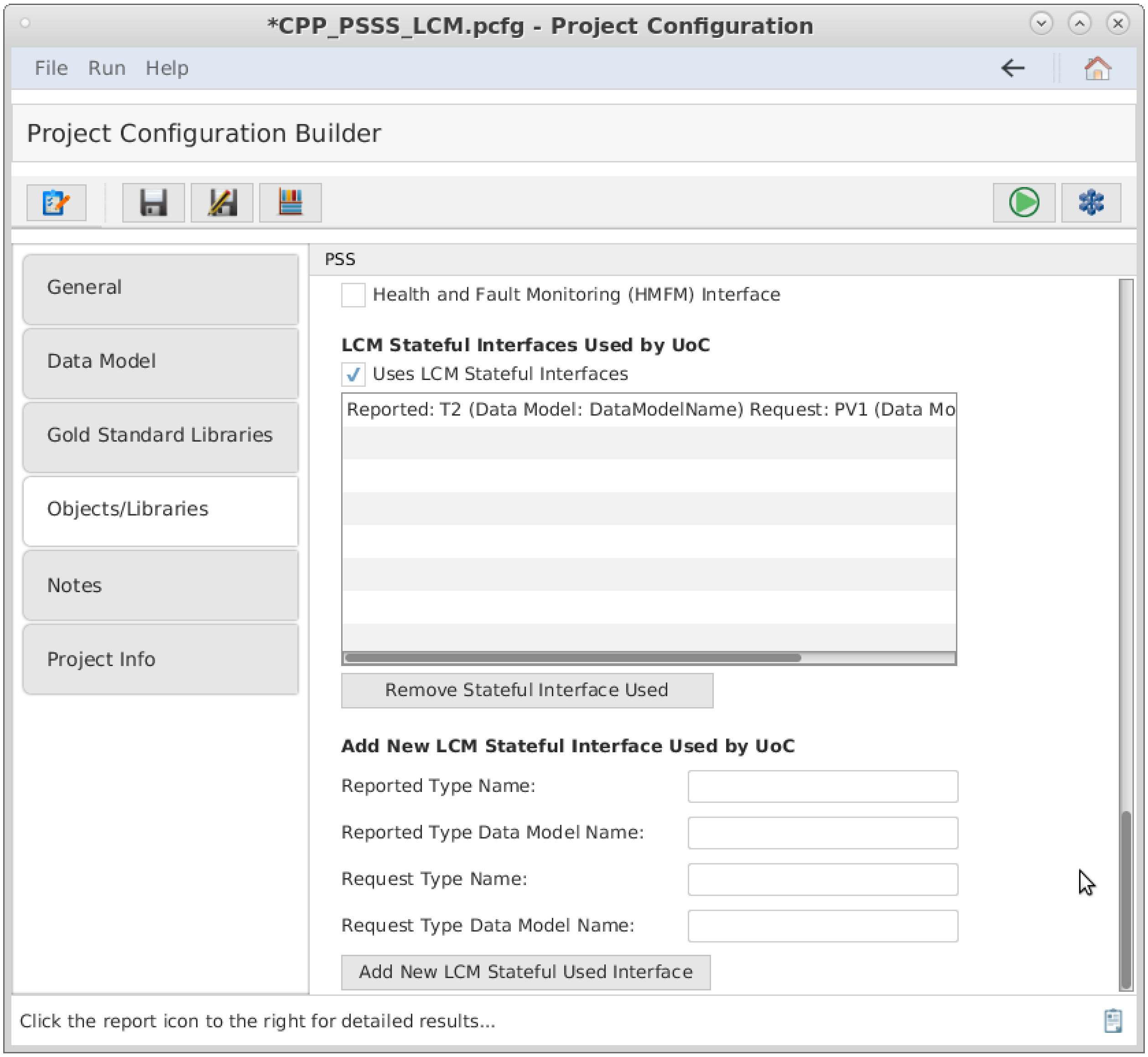
LCM Stateful Interfaces Used by UoC

☑ Uses LCM Stateful Interfaces

Reported: T2 (Data Model: DataModelName) Request: PV1 (Data Mo

Click the report icon to the right for detailed results...

10. Scroll down and enter the information regarding any FACE Life Cycle Management Stateful interfaces that are used by the UoC. If none are used, leave this section blank. Multiple Life Cycle Management Stateful interfaces may be used by a UoC. The required information for each Stateful interface is: the data model and datatype name of the reported datatype, and the data model and datatype name of the request datatype.


*CPP_PSSS_LCM.pcfg - Project Configuration
File
Run
Help
Project Configuration Builder
General
Data Model
Gold Standard Libraries
Objects/Libraries
Notes
Project Info
PSS
Remove Stateful Interface Used
Add New LCM Stateful Used Interface

☐ Health and Fault Monitoring (HMFM) Interface

LCM Stateful Interfaces Used by UoC

☑ Uses LCM Stateful Interfaces

Reported: T2 (Data Model: DataModelName) Request: PV1 (Data Mo

Add New LCM Stateful Interface Used by UoC

Reported Type Name:

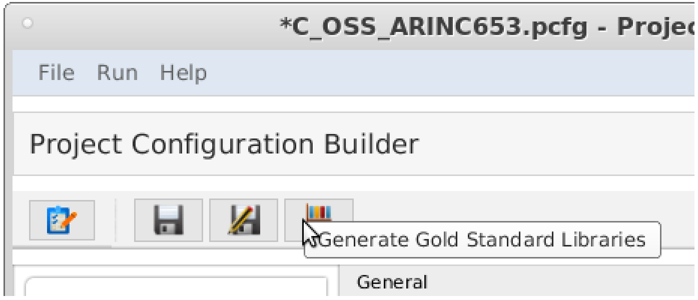
Reported Type Data Model Name:

Request Type Name:

Request Type Data Model Name:

Click the report icon to the right for detailed results...

11. In order to build your source code (assuming you are providing the test object code), you will need FACE interface headers for any interfaces your UoC uses. For example, for a PCS you will need any TSS headers your code uses, as standardized in the FACE standard. The CTS will generate these headers for you. Click the Generate GSLs Button in the upper left corner of the window as shown below to generate the FACE headers as well as the gold standard libraries (GSLs).

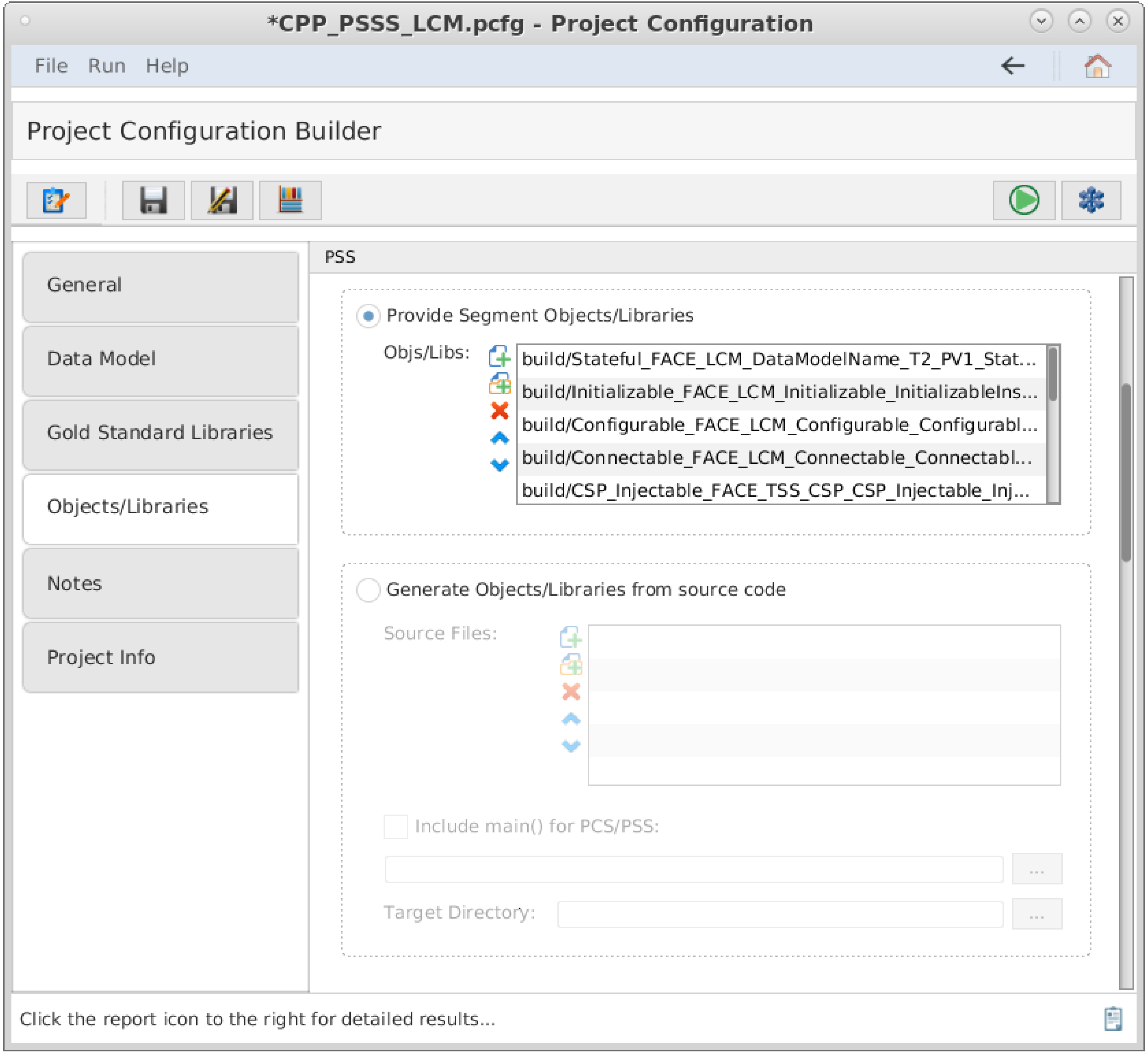

*C_OSS_ARINC653.pcfg - Projec
File
Run
Help
Project Configuration Builder
Generate Gold Standard Libraries
General

12. If providing objects for your UoC, you may now build your objects using the FACE headers generated into the GSL directory's include/FACE subdirectory. Examine the contents of this directory to see the standardized header names of all non-OSS FACE interfaces. Note that for C/C++ all FACE headers should be included with the relative path starting from the FACE directory, for example: "FACE/IOSS/Analog.hpp" The include directories to be used are described in a generated README file in the GSL directory. OSS include directories are listed and are in specific subfolders of the CTS folder goldStandardLibraries.

    Therefore, when building your object code for your UoC, you will need to include these subfolders and include any FACE headers as "FACE/[name of header file]". Note that the headers in this subdirectory are deleted each time the test is run, so do not store any project files in this directory.

The GSL libraries will be generated into the GSL Directory. You may wish to use the GSLs during development to check that your code builds against them, but there is no need to include them in your CTS project. The CTS will rebuild the appropriate GSLs when you run the test and link them as part of the test.

13. Return to the list of segment objects/libraries or source lists as shown below.



● Provide Segment Objects/Libraries

Objs/Libs:
build/Stateful_FACE_LCM_DataModelName_T2_PV1_Stat...
build/Initializable_FACE_LCM_Initializable_InitializableIns...
build/Configurable_FACE_LCM_Configurable_Configurabl...
build/Connectable_FACE_LCM_Connectable_Connectabl...
build/CSP_Injectable_FACE_TSS_CSP_CSP_Injectable_Inj...

○ Generate Objects/Libraries from source code

Click the report icon to the right for detailed results...

14. Enter the full pathnames of your project's object and/or library files. You may add each object file. You may also choose the directory where object files are located. All object files in the directory specified as well as object files in subdirectories will be chosen. Currently, library files must be chosen individually, not by directory. A combination of directories and object/library files may be specified.

15. **Important (applies to all languages):** You must provide a source file that contains the implementation for each of the factory functions that creates an instance of each interface your UoC is providing. To determine which factory functions are necessary, complete the rest of this tutorial and click the "Generate GSLs" button (library icon) in the toolbar. Then, find the generated header/spec file in the generated subfolder 'include/FACE' within the Gold Standard Libraries folder you specified for the project. For C/C++, this file (CTS_Factory_Functions .h or .hpp) will contain the declarations of all expected factory functions that the CTS requires for testing your UoC. You must provide a source file that implements each of these functions. The implementation for each function must instantiate the corresponding concrete class and return a pointer to that object (the return type will be a pointer to the FACE base class). Returning a null pointer is not acceptable. This source file must be provided with your project and will be reviewed to ensure it instantiates your UoC's concrete class for that interface. For Ada, this will generate a spec file (.ads) cts_factory_functions.ads which has the procedures you must implement in the source file. The source file for Ada must be named "cts_factory_functions.adb" and implement each of these procedures, returning a concrete version of each type as an access type. In

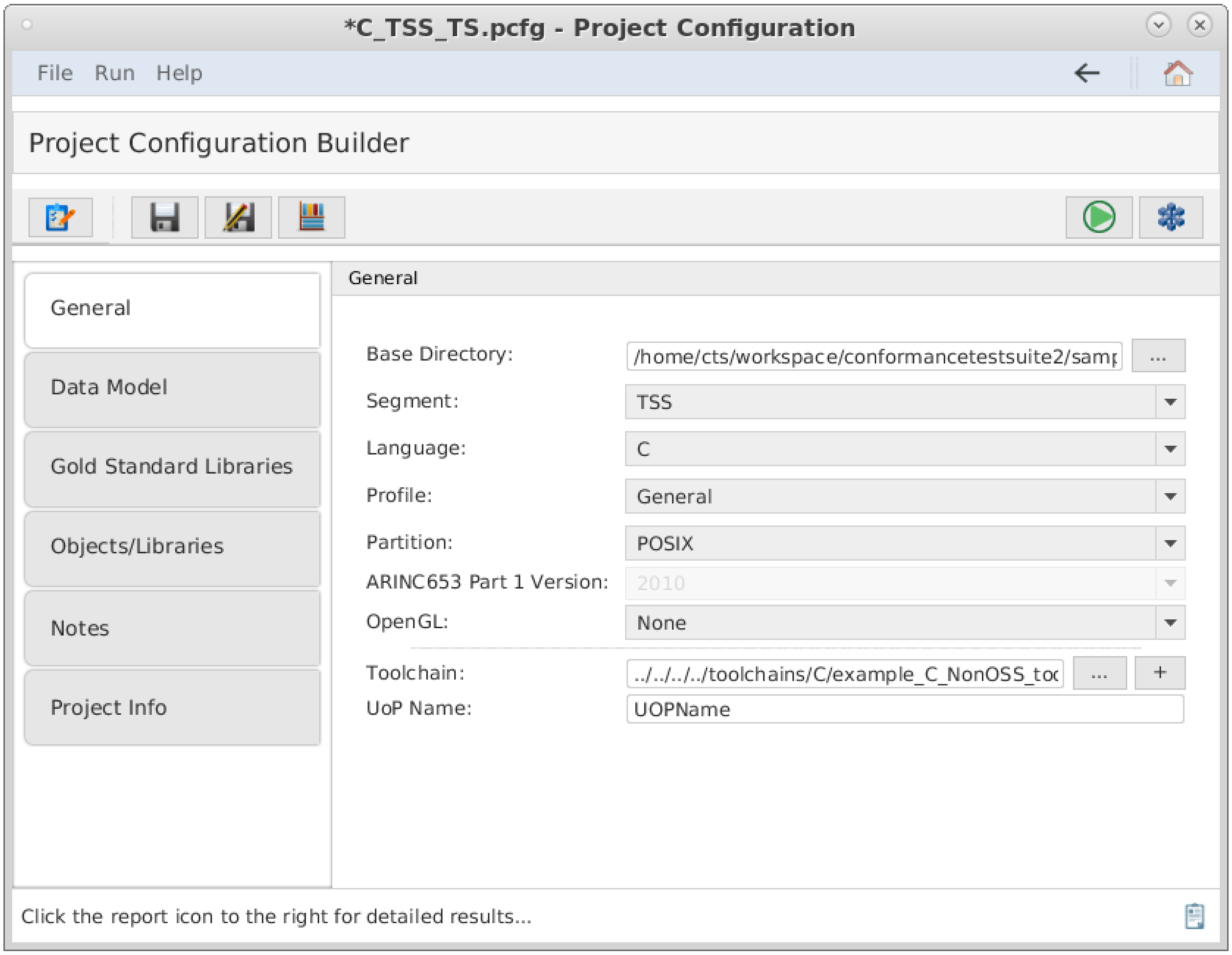
the text field labeled "Source file with factory functions for interfaces provided (C/C++/Ada only)", use the '...' button to the right to browse for the source file. **Any header files included by the source file must exist in one of the Include Paths specified above.** For Java, a source file named CTS_Factory_Functions.java will be generated in the factory/ subfolder (package subfolder) You must fill in the implementation of each function, add any imports, and add this to your project in the CTS in this file field. Note that this file will NOT be overwritten so if the interfaces the UoC uses have changed, you must delete your file in order to regenerate a new one with the new set of functions to implement.

16. Select 📝 to verify that the Project Configuration File is valid.
17. Click the ▶ button at the top of the screen to test the segment. (This may take a few minutes). The results will be written to a PDF file. The directory of the results file will be located in the directory of the project configuration file. This directory path will be listed in the "Output File Location" of the Conformance Test Results page.

Testing a Transport Services Segment (TSS)

What You Must Provide

- Your project's object files (C/C++/Ada) or class files (Java). Alternatively, source files can be provided and the CTS will build them into object or class files using the toolchain configuration.
- Your project's header files (C/C++) or spec files (Ada).
- Your project's data model.
- Your project's toolchain file.



General

Base Directory: /home/cts/workspace/conformancetestsuite2/samp

Segment: TSS

Language: C

Profile: General

Partition: POSIX

ARINC653 Part 1 Version: 2010

OpenGL: None

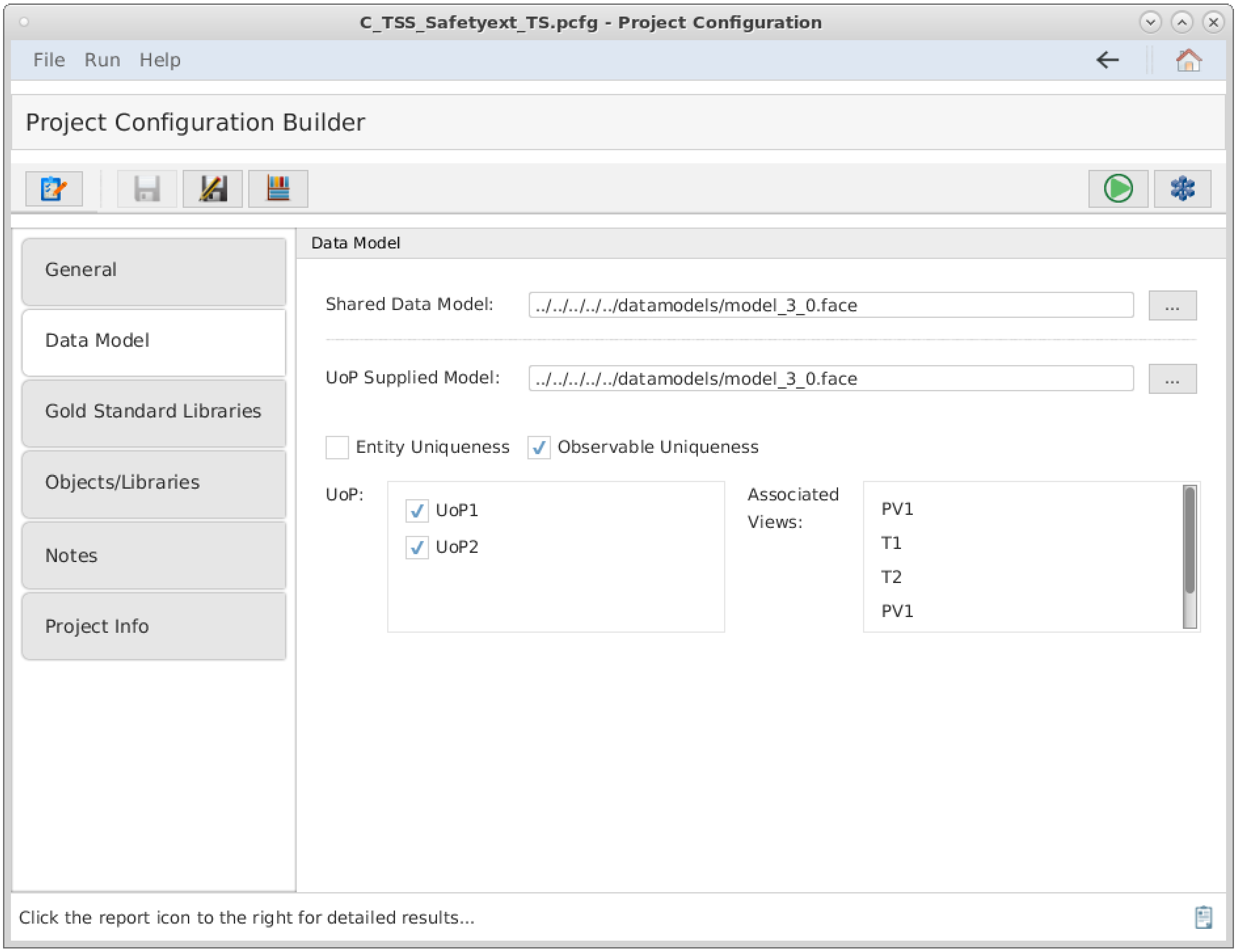
Toolchain: ../../../../toolchains/C/example_C_NonOSS_toc

UoP Name: UOPName

Click the report icon to the right for detailed results...

Procedure

1. Complete the procedure in the *Initialize Conformance Test* section above.
2. Select the Data Model tab to display the data model information below.



C_TSS_Safetyext_TS.pcfg - Project Configuration

Project Configuration Builder

Data Model

Shared Data Model: ../../../../../datamodels/model_3_0.face

UoP Supplied Model: ../../../../../datamodels/model_3_0.face

☐ Entity Uniqueness ☑ Observable Uniqueness

UoP:
☑ UoP1
☑ UoP2

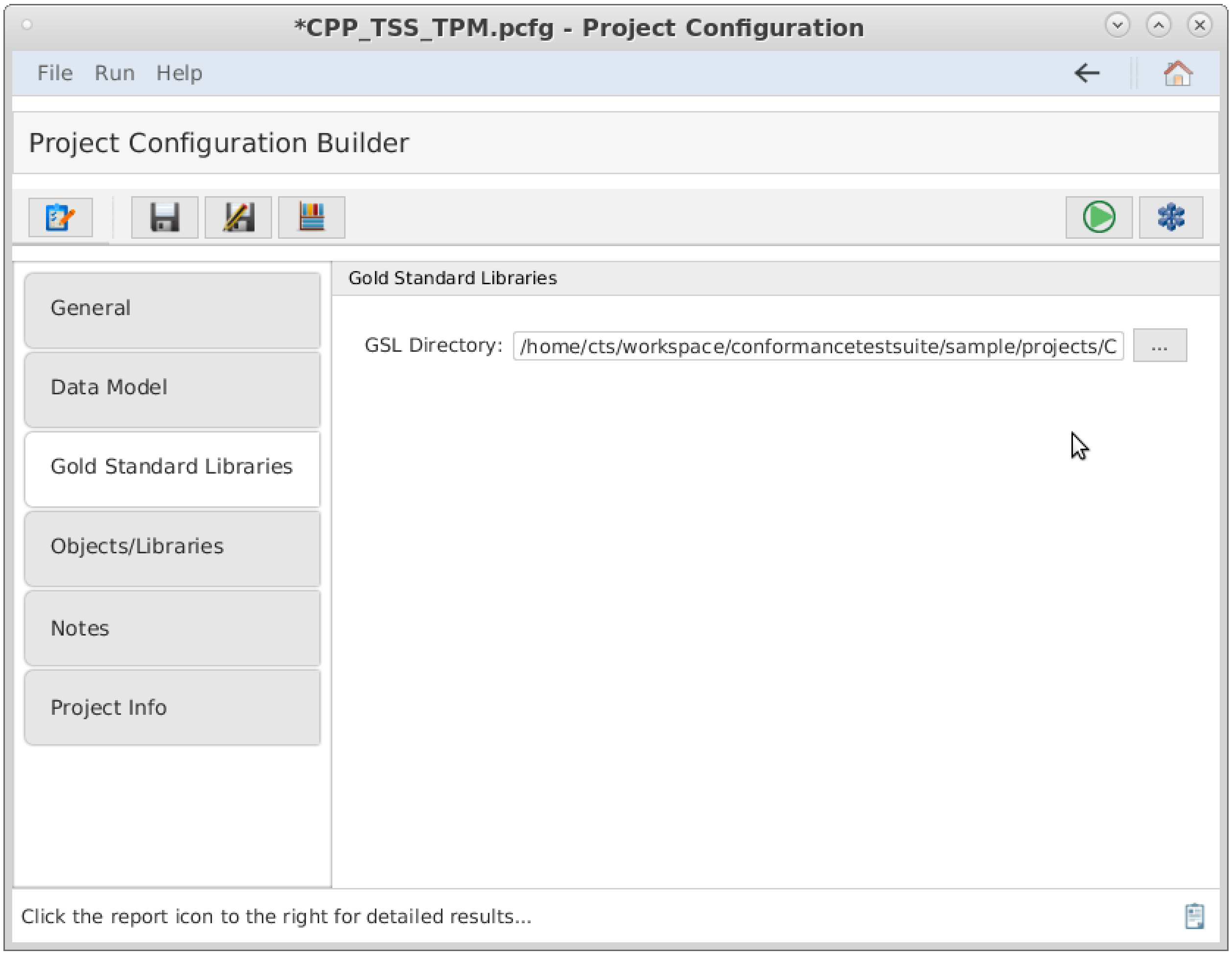
Associated Views:
PV1
T1
T2
PV1

Click the report icon to the right for detailed results...

3. Set the path to the Shared Data Model and UoP Supplied Model. Select the UoPs/UoCs which the TSS services. This determines which datatypes the TS interface supports, if applicable to this project. **Note: This directory is relative to the base directory set in the General tab.**

4. Complete the procedure in the *Testing a Data Model* section below.

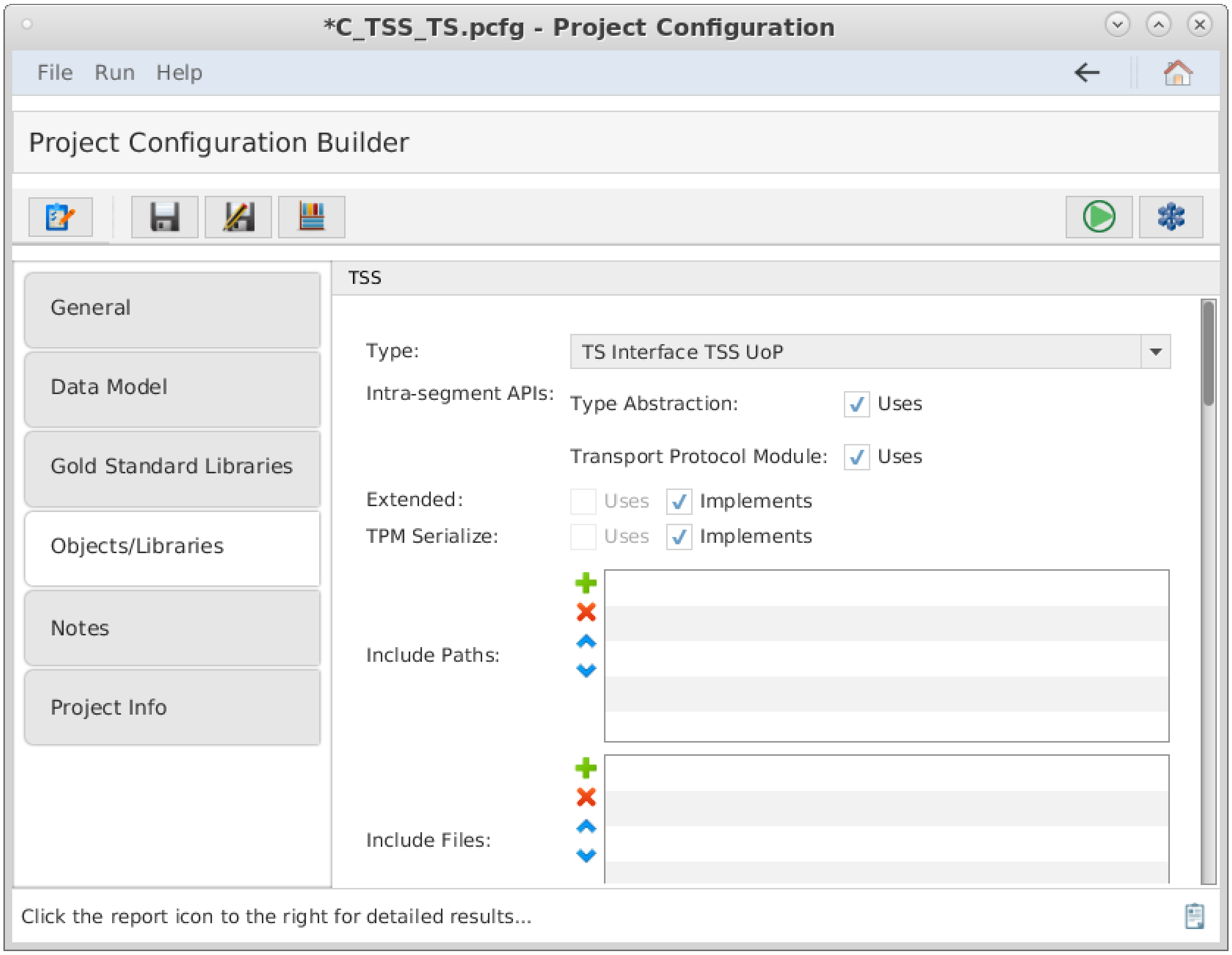
5. Select the Gold Standard Libraries tab to display the options below.



GSL Directory: /home/cts/workspace/conformancetestsuite/sample/projects/C

Click the report icon to the right for detailed results...

6. Set the directory where the gold standard libraries will be generated and stored for the test. **Note: This directory is relative to the base directory set in the General tab.**
7. Select the Objects/Libraries tab to display the transport services information options shown below.
8. Select the UoP Type.
9. Select the Intra-segment APIs if used.
10. For C/C++ projects, use the list boxes to add the include files and include paths (if applicable) for the UoC for the concrete interface implementations provided by the UoC. All 'include files' must exist in one of the 'include paths' specified here. Any files included by the Factory Function Source File (see next few steps in this tutorial) must exist in one of the Include Paths specified.



Type: TS Interface TSS UoP

Intra-segment APIs: Type Abstraction: ☑ Uses

Transport Protocol Module: ☑ Uses

Extended: ☐ Uses ☑ Implements

TPM Serialize: ☐ Uses ☑ Implements

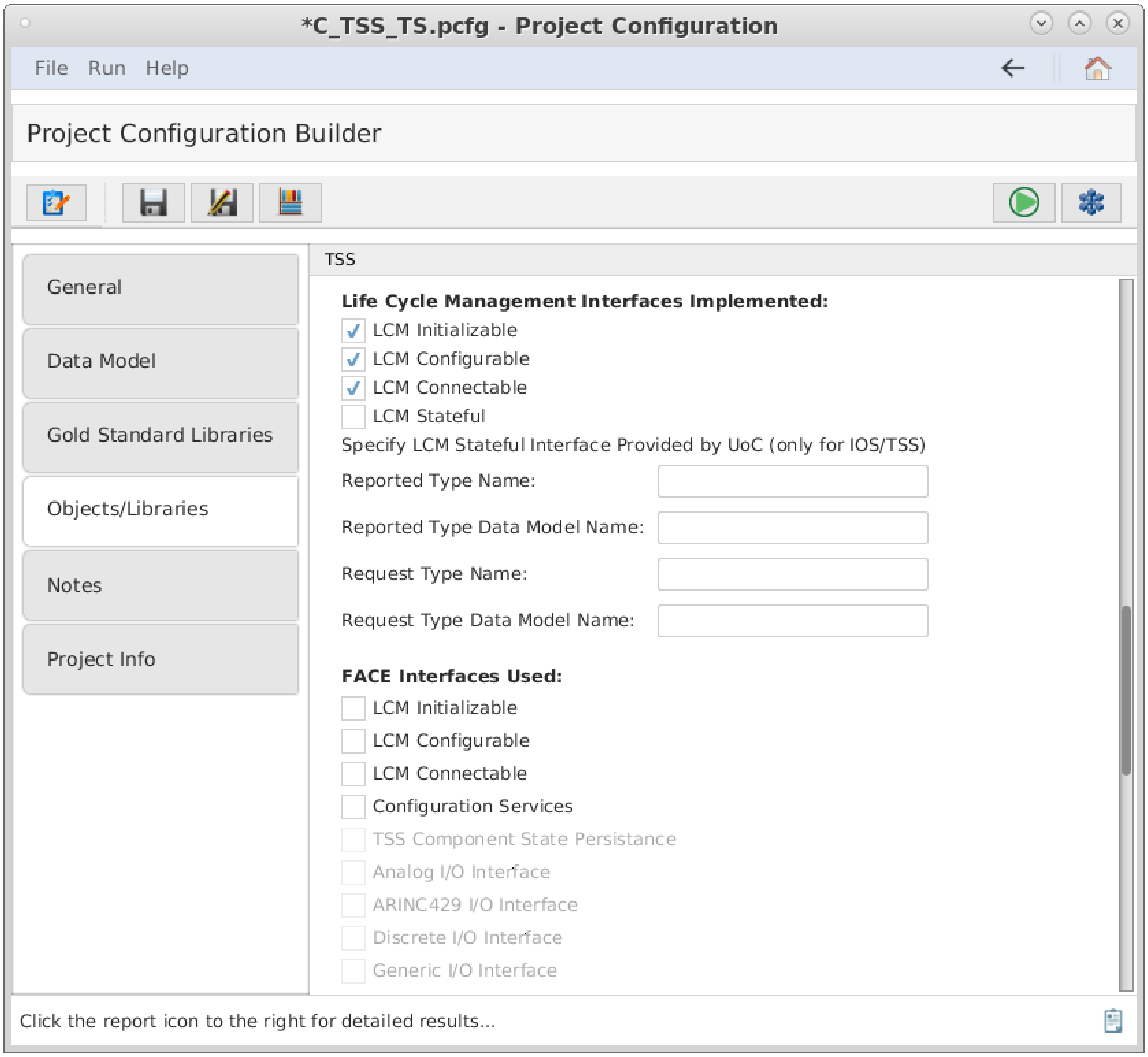
Include Paths:

Include Files:

Click the report icon to the right for detailed results...

11. If providing object files for your UoC: before providing your object or library files, you must build them against the Gold Standard Library headers. The CTS will generate the FACE headers for any interface your UoC uses so that you can build your source code against those. Skip selecting your UoC's object files until you have generated the GSLs and FACE headers and have built your UoC code against these headers. Therefore, skip the section on selecting object files for now.

12. Scroll down and select any of the FACE interfaces the UoC implements. If the Stateful interface is implemented and the project is a IOSS or TSS project, enter the datamodel name and datatype name of the reported and request datatype. For PSSS and PCS projects, the datatypes are defined by the architecture model selected.



Life Cycle Management Interfaces Implemented:

☑ LCM Initializable
☑ LCM Configurable
☑ LCM Connectable
☐ LCM Stateful

Specify LCM Stateful Interface Provided by UoC (only for IOS/TSS)

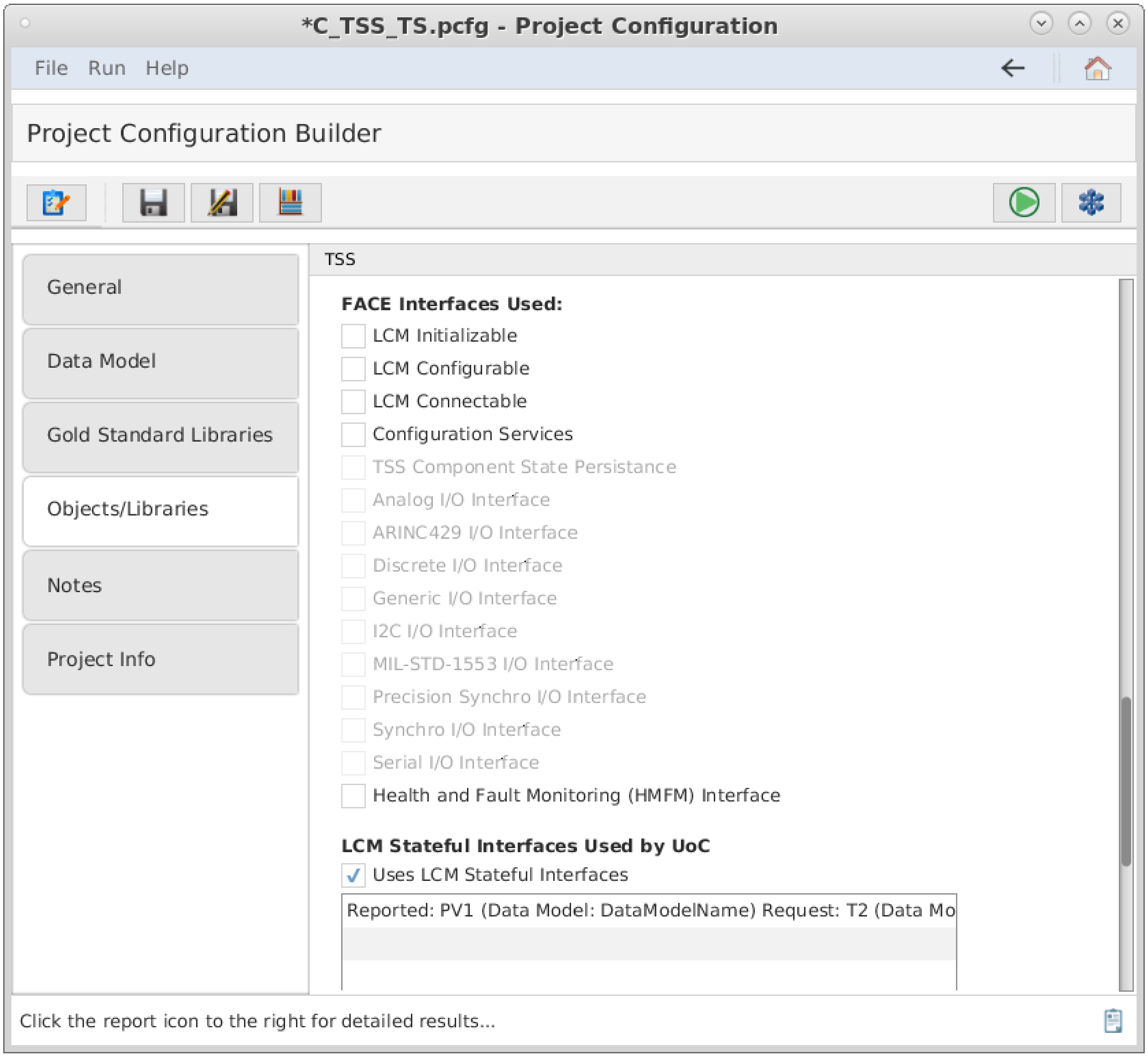
Reported Type Name:
Reported Type Data Model Name:
Request Type Name:
Request Type Data Model Name:

FACE Interfaces Used:

☐ LCM Initializable
☐ LCM Configurable
☐ LCM Connectable
☐ Configuration Services
☐ TSS Component State Persistance
☐ Analog I/O Interface
☐ ARINC429 I/O Interface
☐ Discrete I/O Interface
☐ Generic I/O Interface

Click the report icon to the right for detailed results...

13. Scroll down and select the FACE interfaces that are used by the UoC. The UoC must provide an Injectable interface for each other FACE interface it uses. By specifying that the UoC under test “uses” a given interface, this indicates that it implements an Injectable interface for that interface and this will be tested by the CTS.



FACE Interfaces Used:

- ☐ LCM Initializable
- ☐ LCM Configurable
- ☐ LCM Connectable
- ☐ Configuration Services
- ☐ TSS Component State Persistance
- ☐ Analog I/O Interface
- ☐ ARINC429 I/O Interface
- ☐ Discrete I/O Interface
- ☐ Generic I/O Interface
- ☐ I2C I/O Interface
- ☐ MIL-STD-1553 I/O Interface
- ☐ Precision Synchro I/O Interface
- ☐ Synchro I/O Interface
- ☐ Serial I/O Interface
- ☐ Health and Fault Monitoring (HMFM) Interface

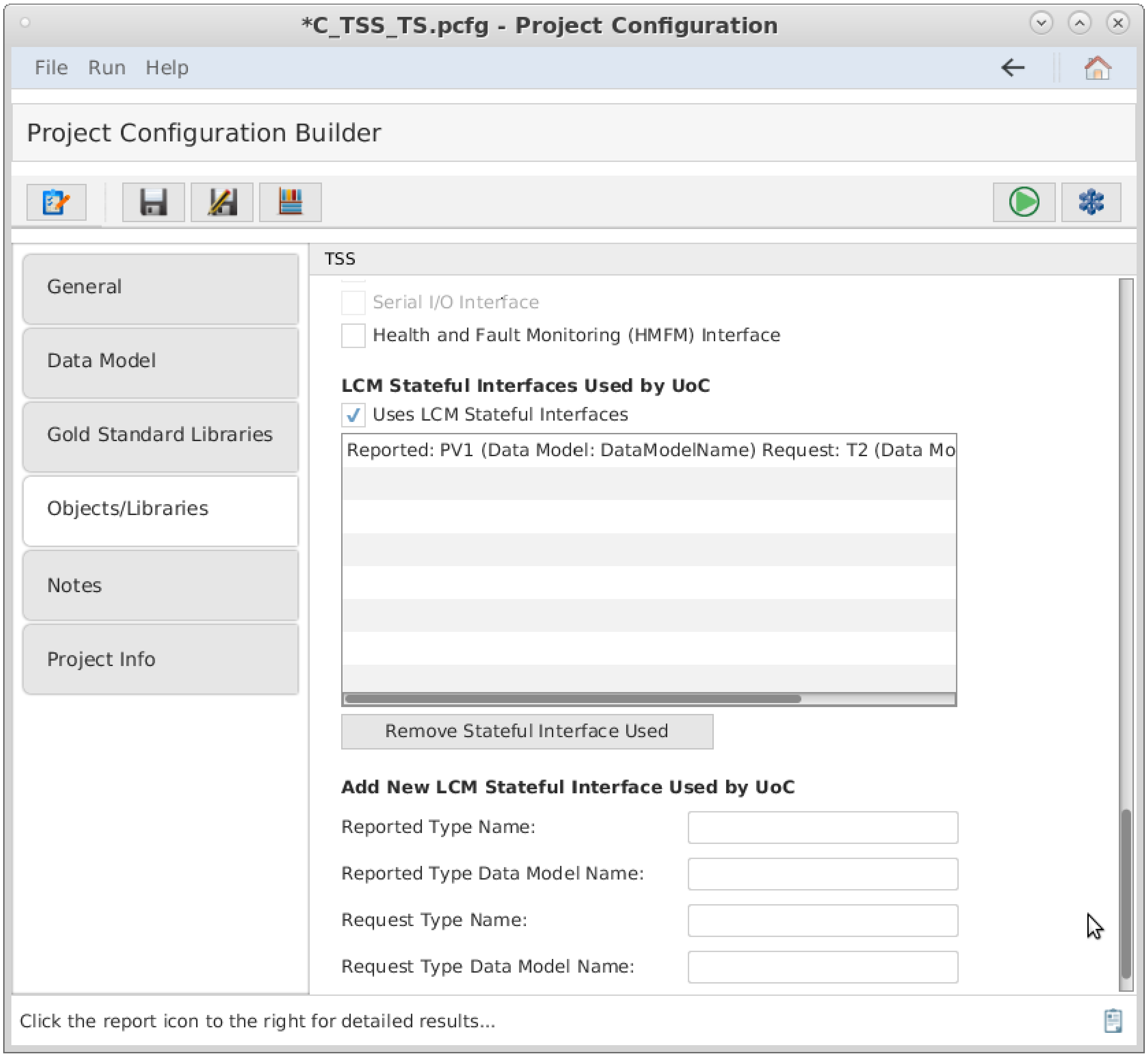
LCM Stateful Interfaces Used by UoC

- ☑ Uses LCM Stateful Interfaces

Reported: PV1 (Data Model: DataModelName) Request: T2 (Data Mo

Click the report icon to the right for detailed results...

14. Scroll down and enter the information regarding any FACE Life Cycle Management Stateful interfaces that are used by the UoC. If none are used, leave this section blank. Multiple Life Cycle Management Stateful interfaces may be used by a UoC. The required information for each Stateful interface is: the data model and datatype name of the reported datatype, and the data model and datatype name of the request datatype.


*C_TSS_TS.pcfg - Project Configuration
File Run Help
Project Configuration Builder
General
Data Model
Gold Standard Libraries
Objects/Libraries
Notes
Project Info
TSS
Remove Stateful Interface Used
Add New LCM Stateful Interface Used by UoC
Reported Type Name:
Reported Type Data Model Name:
Request Type Name:
Request Type Data Model Name:

☐ Serial I/O Interface
☐ Health and Fault Monitoring (HMFM) Interface

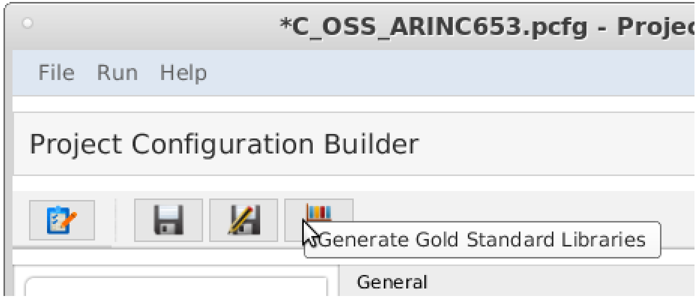
LCM Stateful Interfaces Used by UoC

☑ Uses LCM Stateful Interfaces

Reported: PV1 (Data Model: DataModelName) Request: T2 (Data Mo

Click the report icon to the right for detailed results...

15. In order to build your source code (assuming you are providing the test object code), you will need FACE interface headers for any interfaces your UoC uses. For example, for a PCS you will need any TSS headers your code uses, as standardized in the FACE standard. The CTS will generate these headers for you. Click the Generate GSLs Button in the upper left corner of the window as shown below to generate the FACE headers as well as the gold standard libraries (GSLs).

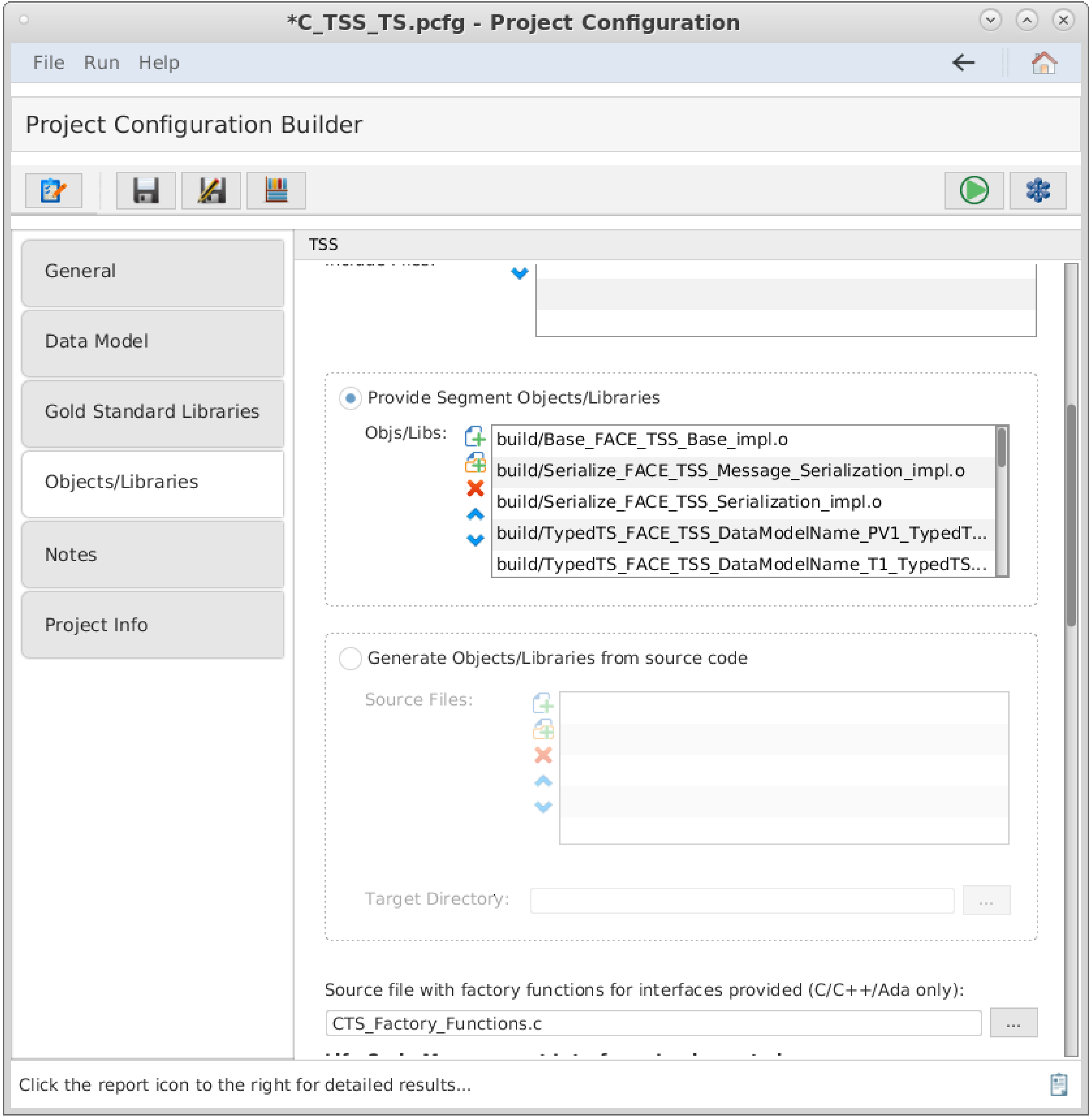

*C_OSS_ARINC653.pcfg - Projec
File Run Help
Project Configuration Builder
Generate Gold Standard Libraries
General

16. If providing objects for your UoC, you may now build your objects using the FACE headers generated into the GSL directory's include/FACE subdirectory and the directories within. Examine the contents of this directory to see the standardized header names of all non-OSS FACE interfaces. Note that for C/C++ all FACE headers should be included with the relative path starting from the FACE directory, for example: "FACE/IOSS/Analog.hpp" The include directories to be used are described in a generated README file in the GSL directory. OSS include directories are listed and are in specific subfolders of the CTS folder goldStandardLibraries.

    Therefore, when building your object code for your UoC, you will need to include these subfolders and include any FACE headers as "FACE/[name of header file]". Note that the headers in this subdirectory are deleted each time the test is run, so do not store any project files in this directory.

The GSL libraries will be generated into the GSL Directory. You may wish to use the GSLs during development to check that your code builds against them, but there is no need to include them in your CTS project. The CTS will rebuild the appropriate GSLs when you run the test and link them as part of the test.

17. Return to the list of segment objects/libraries or source lists as shown below.


*C_TSS_TS.pcfg - Project Configuration
File
Run
Help
Project Configuration Builder
General
Data Model
Gold Standard Libraries
Objects/Libraries
Notes
Project Info
TSS
Source Files:
Target Directory:
...
Click the report icon to the right for detailed results...

● Provide Segment Objects/Libraries

Objs/Libs:
build/Base_FACE_TSS_Base_impl.o
build/Serialize_FACE_TSS_Message_Serialization_impl.o
build/Serialize_FACE_TSS_Serialization_impl.o
build/TypedTS_FACE_TSS_DataModelName_PV1_TypedT...
build/TypedTS_FACE_TSS_DataModelName_T1_TypedTS...

○ Generate Objects/Libraries from source code

Source file with factory functions for interfaces provided (C/C++/Ada only):
CTS_Factory_Functions.c

18. Enter the full pathnames of your project's object and/or library files. You may add each object file. You may also choose the directory where object files are located. All object files in the directory specified as well as object files in subdirectories will be chosen. Currently, library files must be chosen individually, not by directory. A combination of directories and object/library files may be specified.

19. **Important (applies to all languages):** You must provide a source file that contains the implementation for each of the factory functions that creates an instance of each interface your UoC is providing. To determine which factory functions are necessary, complete the rest of this tutorial and click the "Generate GSLs" button (library icon) in the toolbar. Then, find the generated header/spec file in the generated subfolder 'include/FACE' within the Gold Standard Libraries folder you specified for the project. For C/C++, this file (CTS_Factory_Functions .h or .hpp) will contain the declarations of all expected factory functions that the CTS requires for testing your UoC. You must provide a source file that implements each of these functions. The implementation for each

function must instantiate the corresponding concrete class and return a pointer to that object (the return type will be a pointer to the FACE base class). Returning a null pointer is not acceptable. This source file must be provided with your project and will be reviewed to ensure it instantiates your UoC's concrete class for that interface. For Ada, this will generate a spec file (.ads) cts_factory_functions.ads which has the procedures you must implement in the source file. The source file for Ada must be named "cts_factory_functions.adb" and implement each of these procedures, returning a concrete version of each type as an access type. In the text field labeled "Source file with factory functions for interfaces provided (C/C++/Ada only)", use the '...' button to the right to browse for the source file. **Any header files included by the source file must exist in one of the Include Paths specified above.** For Java, a source file named CTS_Factory_Functions.java will be generated in the factory/ subfolder (package subfolder) You must fill in the implementation of each function, add any imports, and add this to your project in the CTS in this file field. Note that this file will NOT be overwritten so if the interfaces the UoC uses have changed, you must delete your file in order to regenerate a new one with the new set of functions to implement.

20. Select 📝 to verify that the Project Configuration File is valid.
21. Click the ▶ button at the top of the screen to test the segment. (This may take a few minutes). The results will be written to a PDF file. The directory of the results file will be located in the directory of the project configuration file. This directory path will be listed in the "Output File Location" of the Conformance Test Results page.

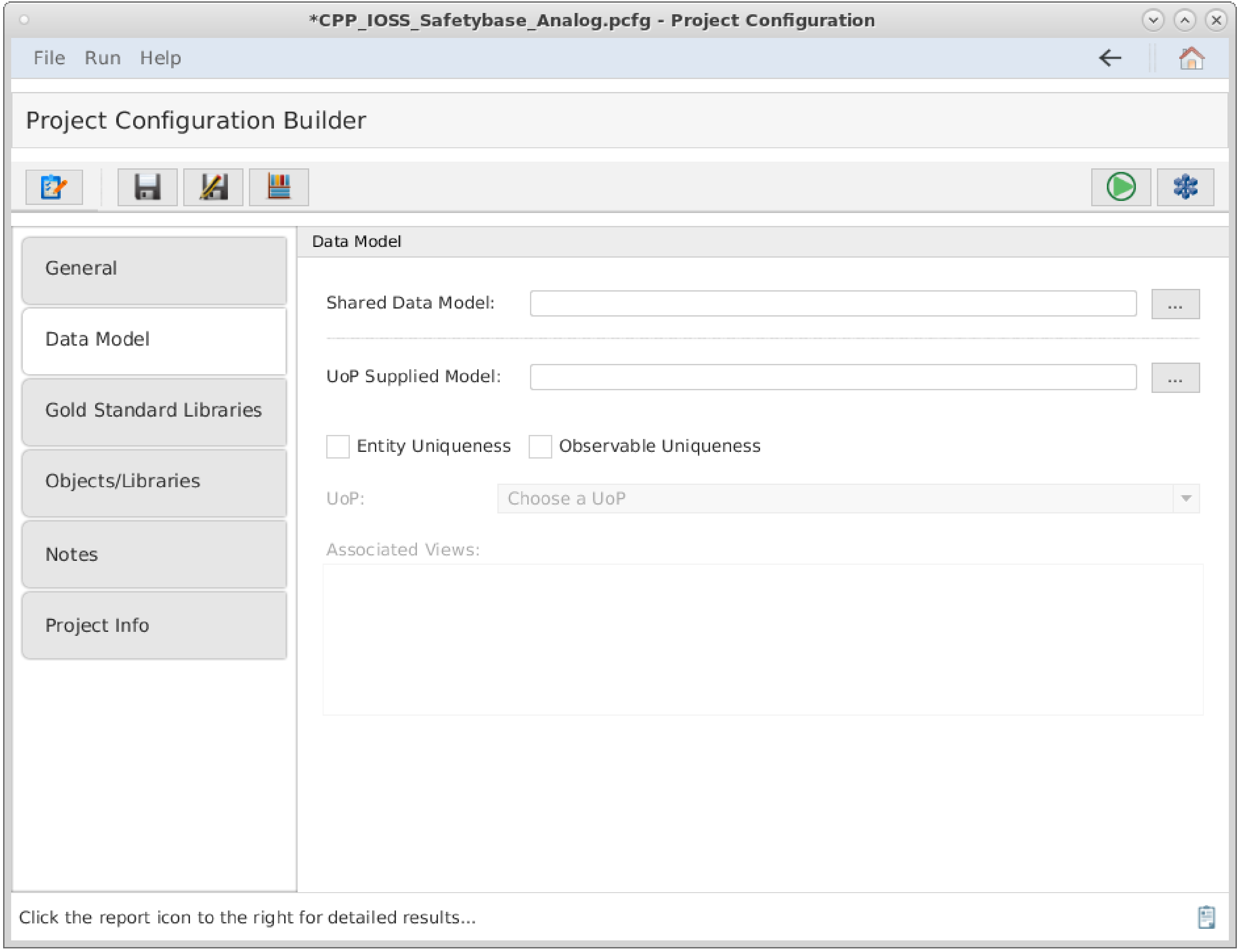
Testing an I/O Services (IOS) Segment

What You Must Provide

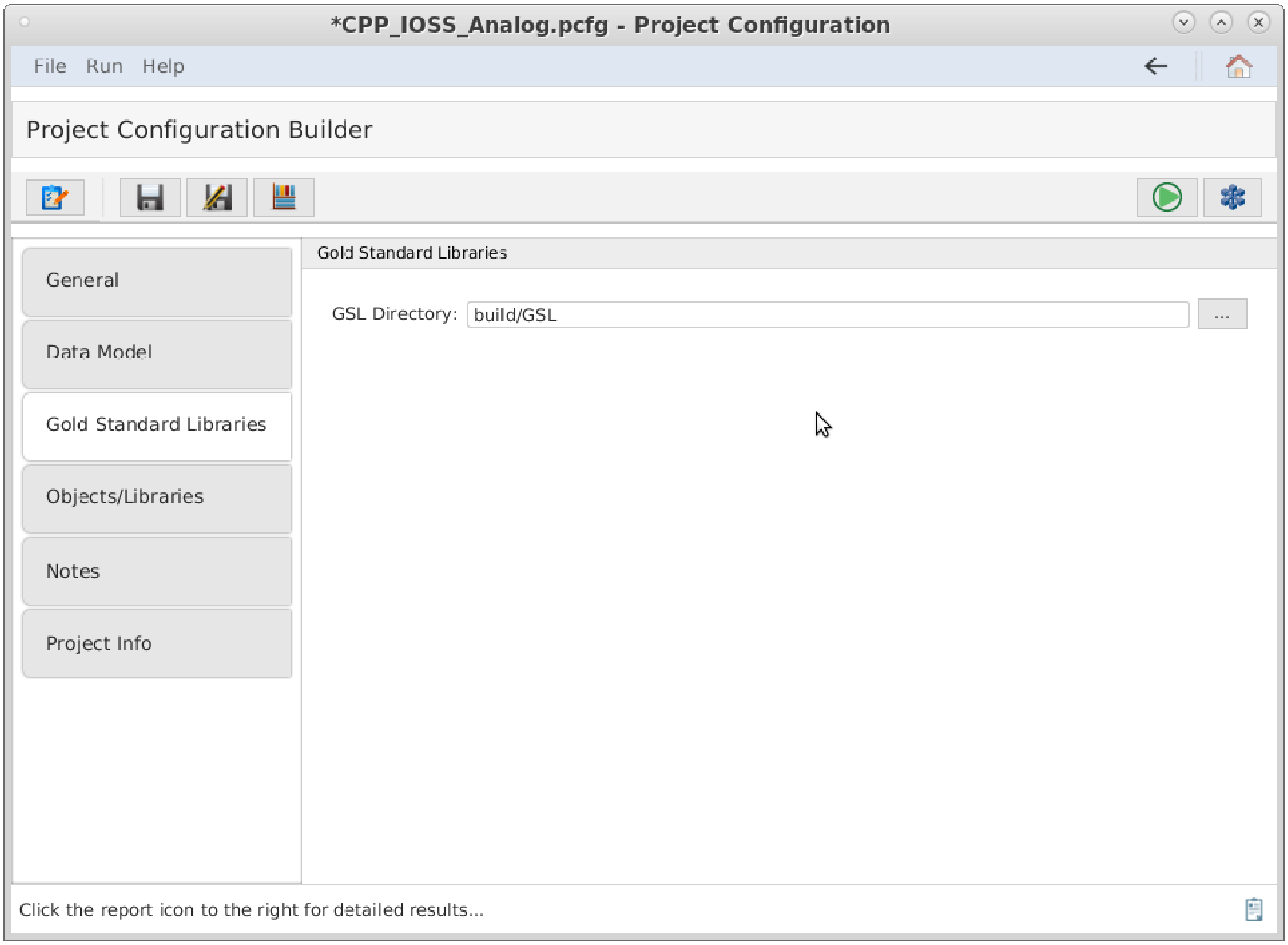
- Your project's object files (C/C++/Ada) or class files (Java). Alternatively, source files can be provided and the CTS will build them into object or class files using the toolchain configuration.
- Your project's header files (C/C++) or spec files (Ada).
- Your project's data model.
- Your project's toolchain file.

Procedure

1. Complete the procedure in the *Initialize Conformance Test* section above.
2. If the IOSS UoC uses or provides a Life Cycle Management Stateful interface select the Data Model tab to display the data model information below and complete these fields.



3. Select the Gold Standard Libraries tab to display the options below.

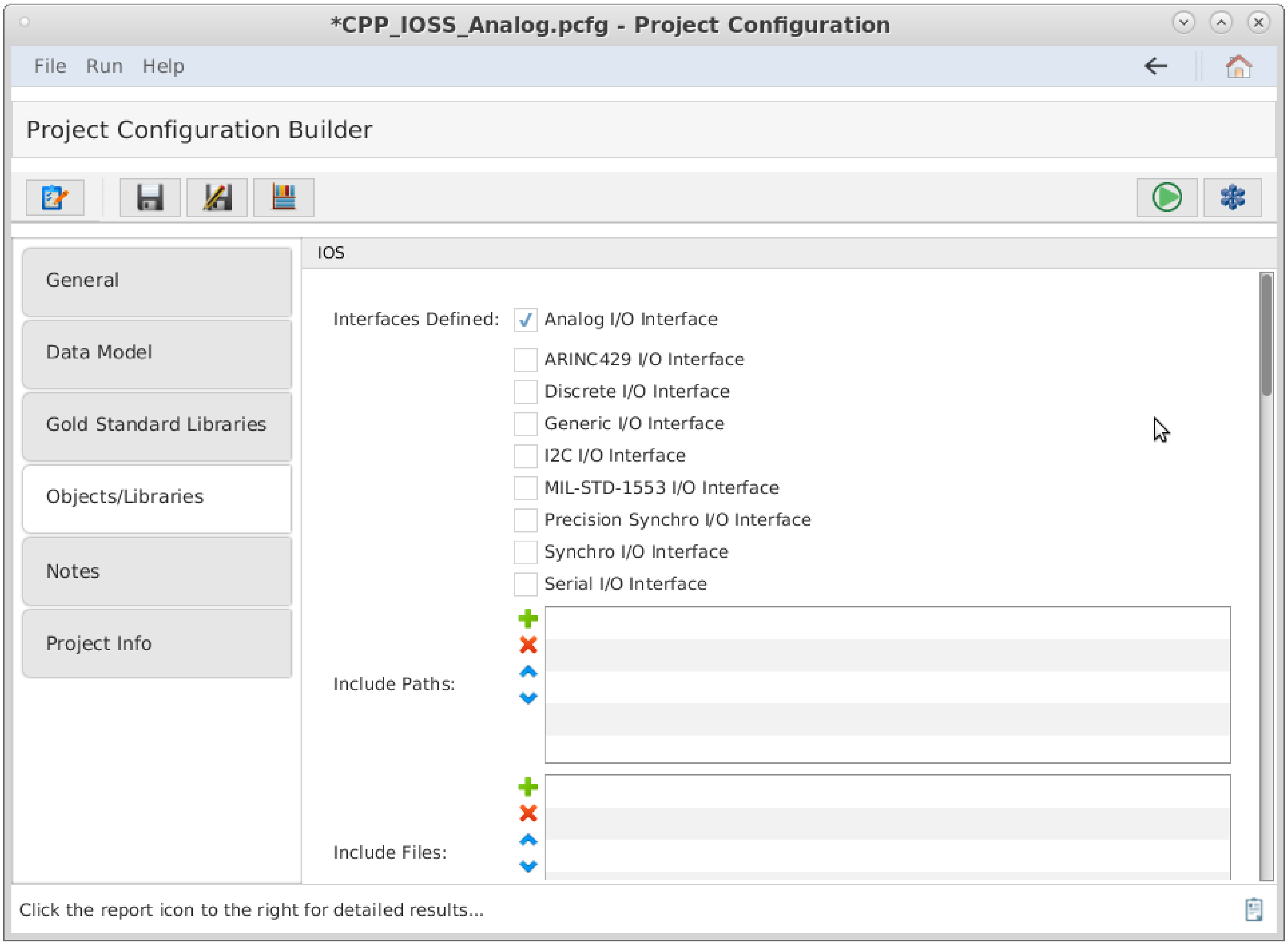


GSL Directory: build/GSL

Click the report icon to the right for detailed results...

4. Set the directory where the gold standard libraries will be generated and stored for the test. **Note: This directory is relative to the base directory set in the General tab.**

5. Select the Objects/Libraries tab to display the IOS services information options shown below.



Interfaces Defined:
- ☑ Analog I/O Interface
- ☐ ARINC429 I/O Interface
- ☐ Discrete I/O Interface
- ☐ Generic I/O Interface
- ☐ I2C I/O Interface
- ☐ MIL-STD-1553 I/O Interface
- ☐ Precision Synchro I/O Interface
- ☐ Synchro I/O Interface
- ☐ Serial I/O Interface

Include Paths:

Include Files:

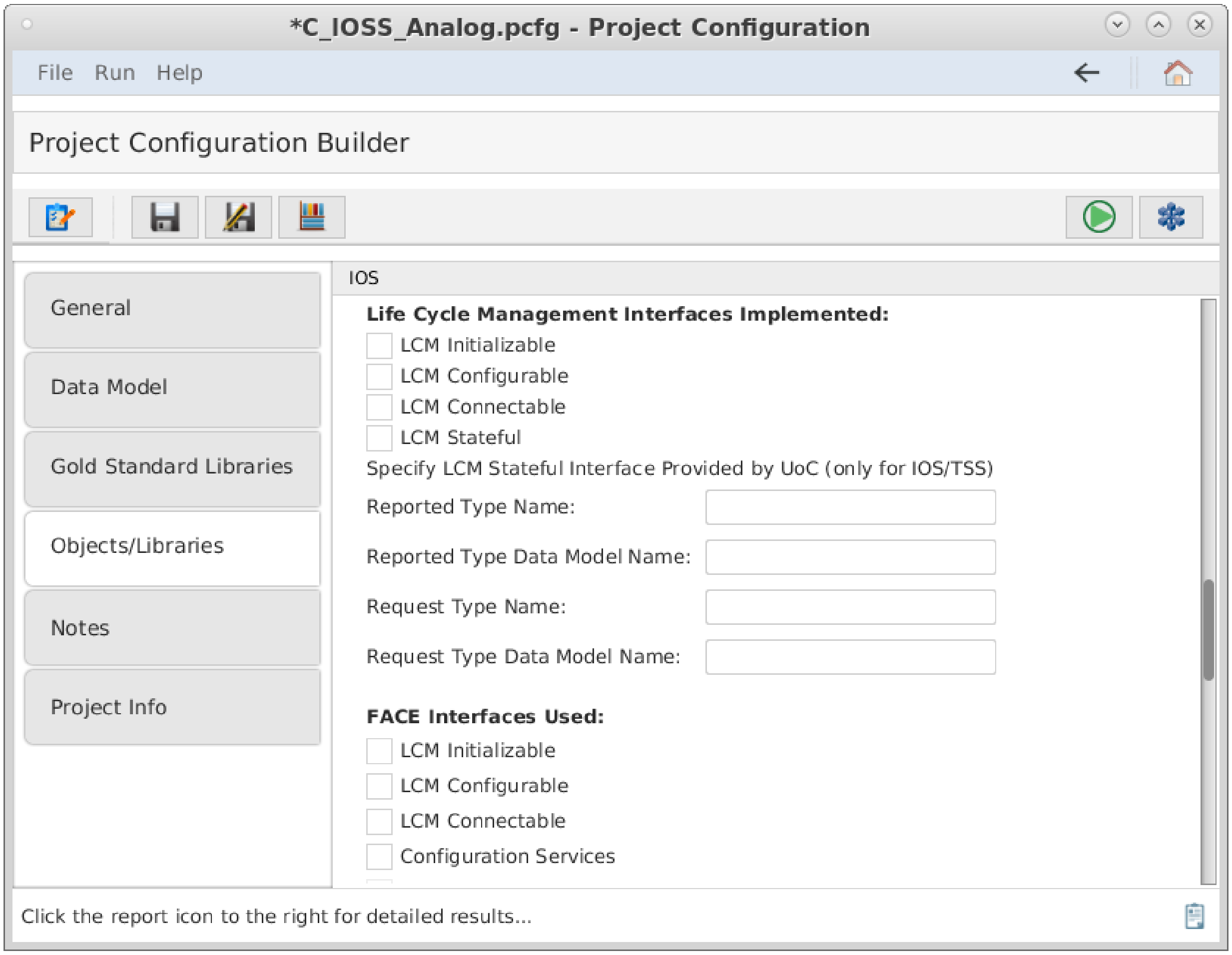
Click the report icon to the right for detailed results...

6. Select the UoC/UoP Type (type of IOSS interface implemented).

7. For C/C++ projects, use the list boxes to add the include files and include paths (if applicable) for the UoC for the concrete interface implementations provided by the UoC. All 'include files' must exist in one of the 'include paths' specified here. Any files included by the Factory Function Source File (see next few steps in this tutorial) must exist in one of the Include Paths specified.

8. If providing object files for your UoC: before providing your object or library files, you must build them against the Gold Standard Library headers. The CTS will generate the FACE headers for any interface your UoC uses so that you can build your source code against those. Skip selecting your UoC's object files until you have generated the GSLs and FACE headers and have built your UoC code against these headers. Therefore, skip the section on selecting object files for now.

9. Scroll down and select any of the FACE interfaces the UoC implements. If the Stateful interface is implemented and the project is a IOSS or TSS project, enter the datamodel name and datatype name of the reported and request datatype. For PSSS and PCS projects, the datatypes are defined by the architecture model selected.



Life Cycle Management Interfaces Implemented:

☐ LCM Initializable
☐ LCM Configurable
☐ LCM Connectable
☐ LCM Stateful

Specify LCM Stateful Interface Provided by UoC (only for IOS/TSS)

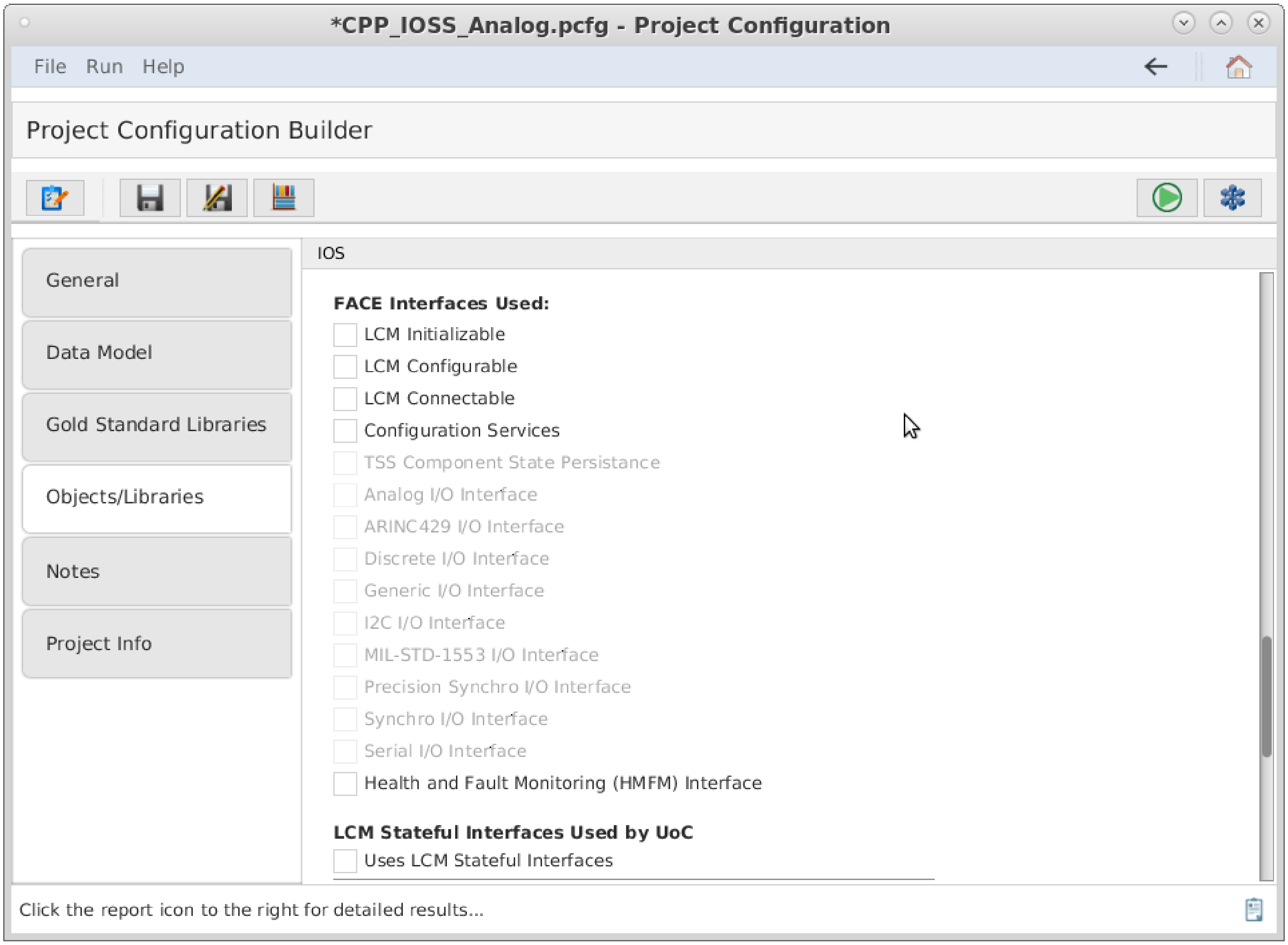
Reported Type Name:
Reported Type Data Model Name:
Request Type Name:
Request Type Data Model Name:

FACE Interfaces Used:

☐ LCM Initializable
☐ LCM Configurable
☐ LCM Connectable
☐ Configuration Services

Click the report icon to the right for detailed results...

10. Scroll down and select the FACE interfaces that are used by the UoC. The UoC must provide an Injectable interface for each FACE interface it uses. By specifying that the UoC under test “uses” a given interface, this indicates that it implements an Injectable interface for that interface and this will be tested by the CTS.



FACE Interfaces Used:

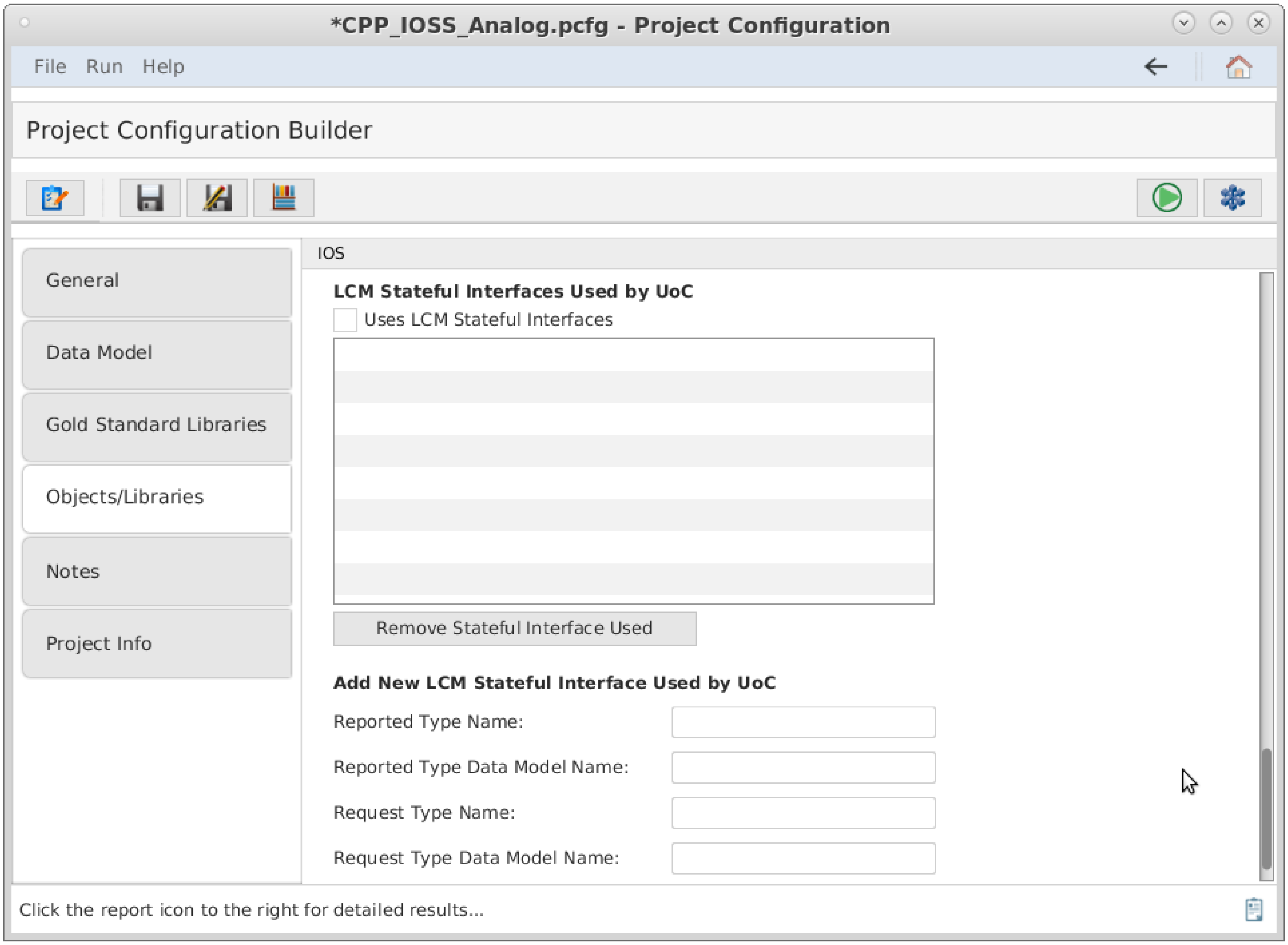
☐ LCM Initializable
☐ LCM Configurable
☐ LCM Connectable
☐ Configuration Services
☐ TSS Component State Persistance
☐ Analog I/O Interface
☐ ARINC429 I/O Interface
☐ Discrete I/O Interface
☐ Generic I/O Interface
☐ I2C I/O Interface
☐ MIL-STD-1553 I/O Interface
☐ Precision Synchro I/O Interface
☐ Synchro I/O Interface
☐ Serial I/O Interface
☐ Health and Fault Monitoring (HMFM) Interface

LCM Stateful Interfaces Used by UoC

☐ Uses LCM Stateful Interfaces

Click the report icon to the right for detailed results...

11. Scroll down and enter the information regarding any FACE Life Cycle Management Stateful interfaces that are used by the UoC. If none are used, leave this section blank. Multiple Life Cycle Management Stateful interfaces may be used by a UoC. The required information for each Stateful interface is: the data model and datatype name of the reported datatype, and the data model and datatype name of the request datatype.


*CPP_IOSS_Analog.pcfg - Project Configuration
File Run Help
Project Configuration Builder
General
Data Model
Gold Standard Libraries
Objects/Libraries
Notes
Project Info
IOS
Remove Stateful Interface Used

LCM Stateful Interfaces Used by UoC

☐ Uses LCM Stateful Interfaces

Add New LCM Stateful Interface Used by UoC

Reported Type Name:

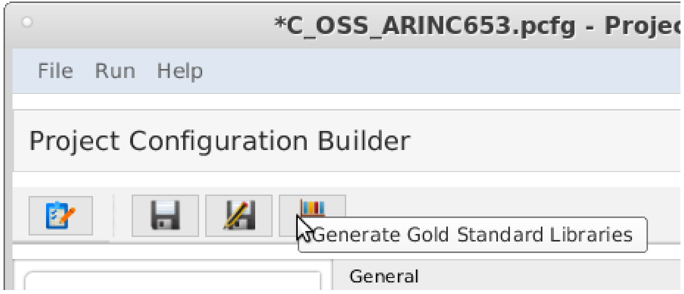
Reported Type Data Model Name:

Request Type Name:

Request Type Data Model Name:

Click the report icon to the right for detailed results...

12. In order to build your source code (assuming you are providing the test object code), you will need FACE interface headers for any interfaces your UoC uses. For example, for a PCS you will need any TSS headers your code uses, as standardized in the FACE standard. The CTS will generate these headers for you. Click the Generate GSLs Button in the upper left corner of the window as shown below to generate the FACE headers as well as the gold standard libraries (GSLs).

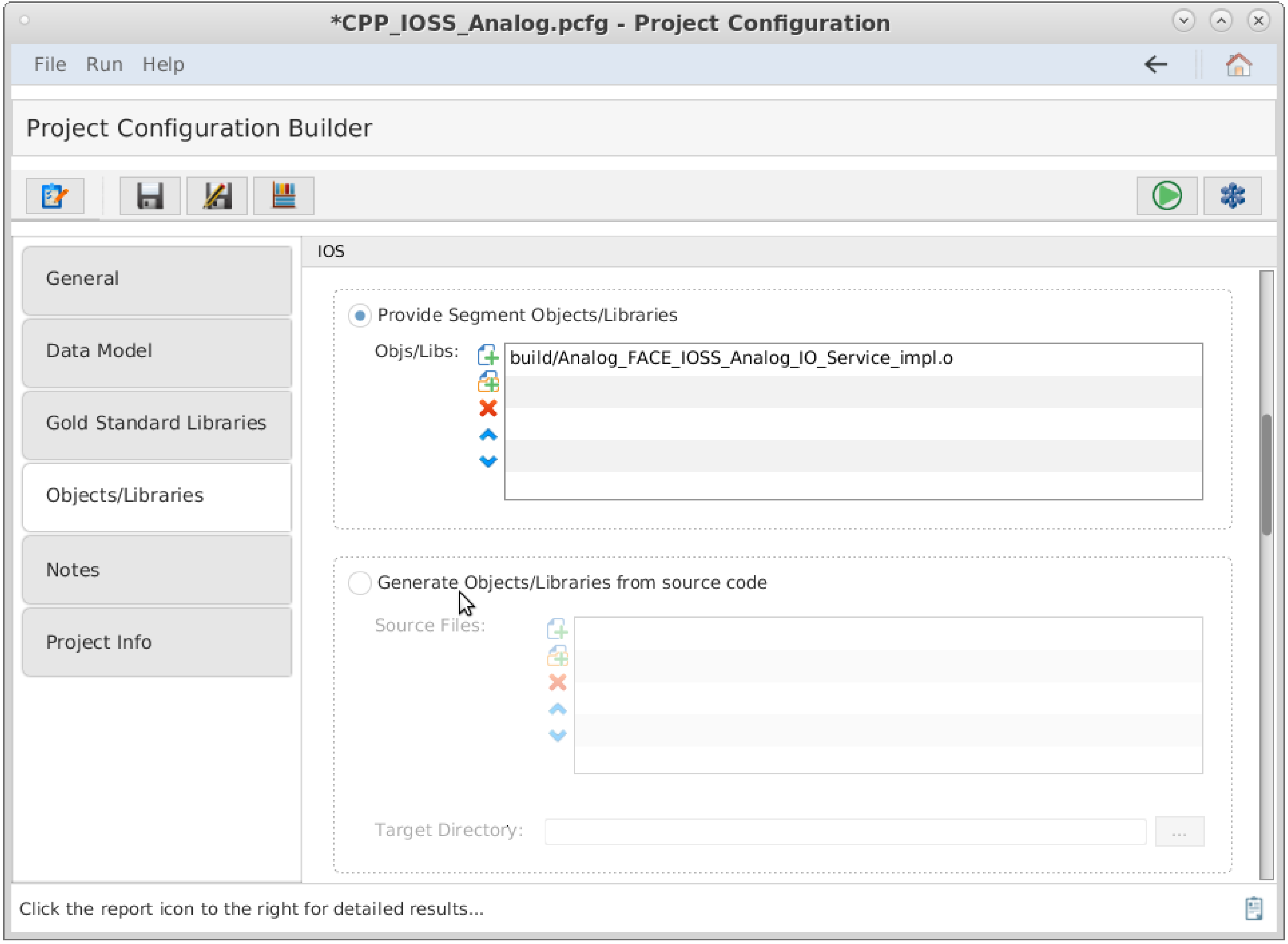

*C_OSS_ARINC653.pcfg - Projec
File Run Help
Project Configuration Builder
Generate Gold Standard Libraries
General

13. If providing objects for your UoC, you may now build your objects using the FACE headers generated into the GSL directory's include/FACE subdirectory. Examine the contents of this directory to see the standardized header names of all non-OSS FACE interfaces. Note that for C/C++ all FACE headers should be included with the relative path starting from the FACE directory, for example: "FACE/IOSS/Analog.hpp" The include directories to be used are described in a generated README file in the GSL directory. OSS include directories are listed and are in specific subfolders of the CTS folder goldStandardLibraries.

    Therefore, when building your object code for your UoC, you will need to include these subfolders and include any FACE headers as "FACE/[name of header file]". Note that the headers in this subdirectory are deleted each time the test is run, so do not store any project files in this directory.

    The GSL libraries will be generated into the GSL Directory. You may wish to use the GSLs during development to check that your code builds against them, but there is no need to include them in your CTS project. The CTS will rebuild the appropriate GSLs when you run the test and link them as part of the test.

14. Return to the segment objects/libraries or source lists as shown below.



● Provide Segment Objects/Libraries

Objs/Libs: build/Analog_FACE_IOSS_Analog_IO_Service_impl.o

○ Generate Objects/Libraries from source code

Source Files:

Target Directory:

Click the report icon to the right for detailed results...

15. Enter the full pathnames your project's object and/or library files. You may add each object file. You may also choose the directory where object files are located. All object files in the directory specified as well as object files in subdirectories will be chosen. Currently, library files must be chosen individually, not by directory. A combination of directories and object/library files may be specified.

16. **Important (applies to all languages):** You must provide a source file that contains the implementation for each of the factory functions that creates an instance of each interface your UoC is providing. To determine which factory functions are necessary, complete the rest of this tutorial and click the "Generate GSLs" button (library icon) in the toolbar. Then, find the generated header/spec file in the generated subfolder 'include/FACE' within the Gold Standard Libraries folder you specified for the project. For C/C++, this file (CTS_Factory_Functions .h or .hpp) will contain the declarations of all expected factory functions that the CTS requires for testing your UoC. You must provide a source file that implements each of these functions. The implementation for each function must instantiate the corresponding concrete class and return a pointer to that object (the return type will be a pointer to the FACE base class). Returning a null pointer is not acceptable. This source file must be provided with your project and will be reviewed to ensure it instantiates your UoC's concrete class for that interface. For Ada, this will generate a spec file (.ads) cts_factory_functions.ads which has the procedures you must implement in the source file. The source file for Ada must be named "cts_factory_functions.adb" and implement each of these procedures, returning a concrete version of each type as an access type. In the text field labeled "Source file with factory functions for interfaces provided (C/C++/Ada only)", use the '...' button to the right to browse for the source file. **Any header files included by the source file must exist in one of the Include Paths specified above.**
    For Java, a source file named CTS_Factory_Functions.java will be generated in the factory/ subfolder (package subfolder) You must fill in the implementation of each function, add any imports, and add this to your project in the CTS in this file field. Note that this file will NOT be overwritten so if the interfaces the UoC uses have changed, you must delete your file in order to regenerate a new one with the new set of functions to implement.

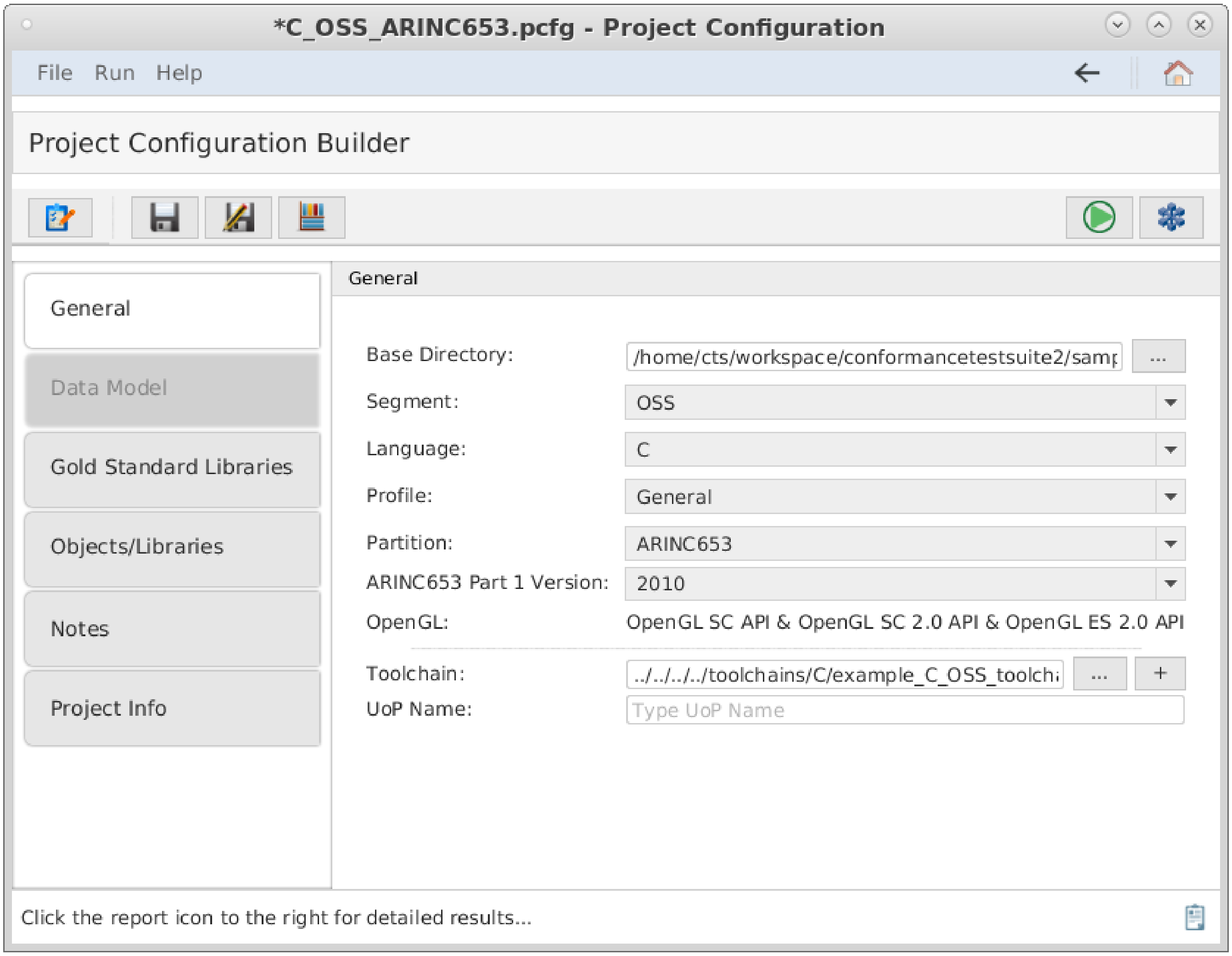
17. Select to verify that the Project Configuration File is valid.

18. Click the ▶ button at the top of the screen to test the segment. (This may take a few minutes). The results will be displayed in a PDF file. To open the PDF click the "open results" button. The directory of the PDF results will be located in the directory of the project configuration file. This directory path will be listed in the "Output File Location" of the Conformance Test Results page.

Testing an Operating System (OSS) Segment

What You Must Provide

- Your target OS's include path.
- Your target OS's object files.



Base Directory: /home/cts/workspace/conformancetestsuite2/samp

Segment: OSS

Language: C

Profile: General

Partition: ARINC653

ARINC653 Part 1 Version: 2010

OpenGL: OpenGL SC API & OpenGL SC 2.0 API & OpenGL ES 2.0 API

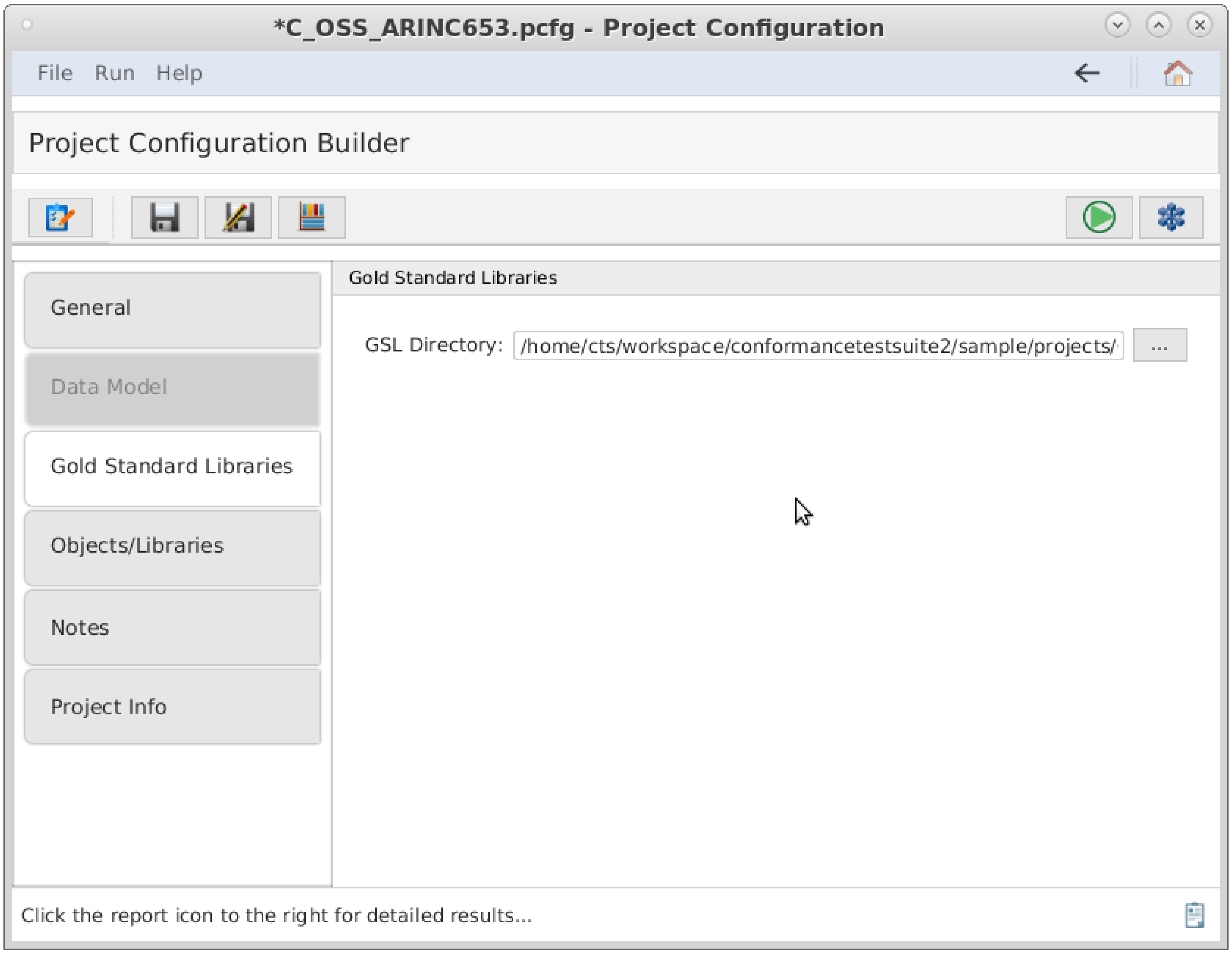
Toolchain: ../../../../toolchains/C/example_C_OSS_toolchi

UoP Name: Type UoP Name

Click the report icon to the right for detailed results...

Procedure

1. Complete the procedure in the *Initialize Conformance Test* section above.
2. Select the Gold Standard Libraries tab to display the options below.

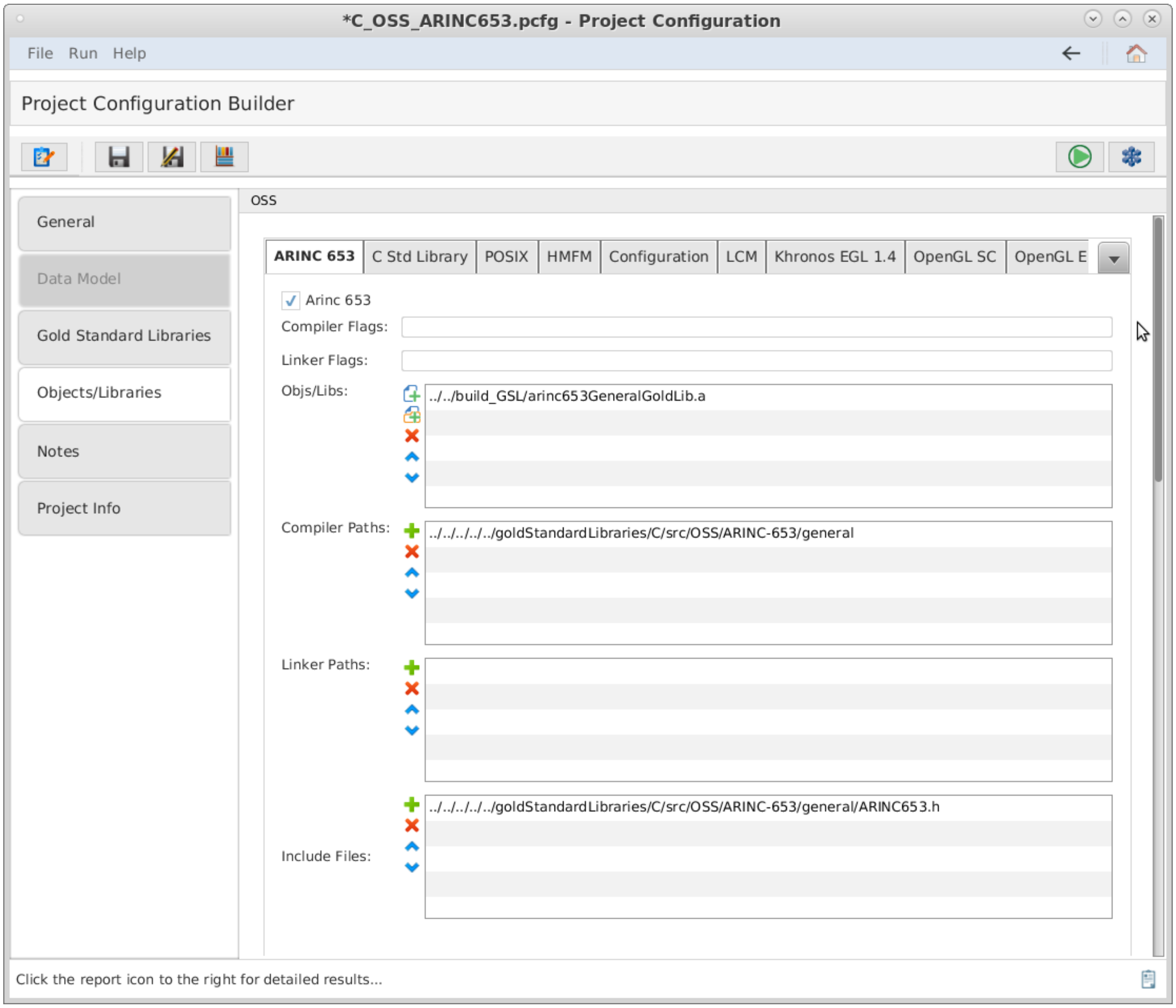


GSL Directory: /home/cts/workspace/conformancetestsuite2/sample/projects/

Click the report icon to the right for detailed results...

3. Set the directory where the gold standard libraries will be generated and stored for the test. **Note: This directory is relative to the base directory set in the General tab.**

4. Select the Objects/Libraries tab to display the operating system information below.



☑ Arinc 653

Compiler Flags:

Linker Flags:

Objs/Libs: ../../build_GSL/arinc653GeneralGoldLib.a

Compiler Paths: ../../../../../goldStandardLibraries/C/src/OSS/ARINC-653/general

Linker Paths:

Include Files: ../../../../../goldStandardLibraries/C/src/OSS/ARINC-653/general/ARINC653.h

Click the report icon to the right for detailed results...

Notice the check-mark on the OSS tab and each valid API test check options are now enabled. Valid APIs depend on Language and Profile selections. See the table below for more details.

Language/Profile	ARINC 653	C Std Library	C++ Std Library	HMFM	Java	Khronus Group EGL 1.4	OpenGL ES 2.0	OpenGL SC 2.0	POSIX	Configuration	LCM
C/GP	X	X		X		X	X		X	X	X
C/SB, C/SE	X	X		X				X	X	X	X
C/S	X	X		X					X	X	X
C++/All			X	X						X	X
Ada/All	X			X						X	X
Java/GP					X					X	X

Note: GP=General Purpose, SB=Safety Base, SE=Safety Extended, S=Security, All=All profiles

Table 1. OSS Tests based on language and profile

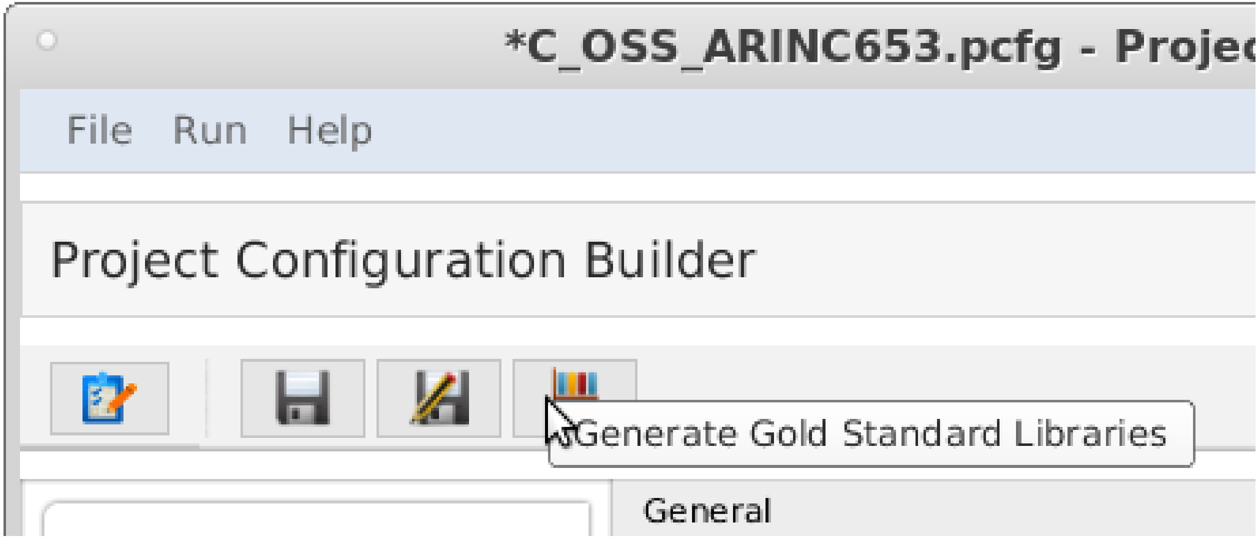
4. Check each OS API you wish to test. Notice their options are now editable.
5. For each OS API under test, place any specific compiler flags that are needed. Note: General compiler flags can be specified under the Build tab. These flags should be unique to the OS API under test.

6. For each OS API under test, place any specific linker flags that are needed. Note: General linker flags can be specified under the Build tab. These flags should be unique to the OS API under test.

7. For each OS API under test, enter any directories that should be in the include path for each OS API interface.

8. For each OS API under test, enter the full pathnames to any header files associated with the interface. These files must be located in one of the directories specified under 'compiler paths' (include paths). Any files included by the Factory Function Source File (see next few steps in this tutorial) must exist in one of the Include Paths specified.

9. **For providing the FACE Configuration interface only:** If providing object files for your UoC: before providing your object or library files, you must build them against the Gold Standard Library headers. The CTS will generate the FACE Configuration headers so that you can build your source code against them. Skip selecting your UoC's object files until you have generated the GSLs and FACE headers and have built your UoC code against these headers. Therefore, skip the section on selecting object files for now.


*C_OSS_ARINC653.pcfg - Projec
File
Run
Help
Project Configuration Builder
Generate Gold Standard Libraries
General

10. **For providing the FACE Configuration interface only:** for providing objects for your UoC, you may now build your objects using the FACE headers generated into the GSL directory's include/FACE subdirectory. Examine the contents of this directory to see the standardized header names of all non-OSS FACE interfaces. Note that for C/C++ all FACE headers should be included with the relative path starting from the FACE directory, for example: "FACE/IOSS/Analog.hpp" The include directories to be used are described in a generated README file in the GSL directory. OSS include directories are listed and are in specific subfolders of the CTS folder goldStandardLibraries.

    Therefore, when building your object code for your UoC, you will need to include these subfolders and include any FACE headers as "FACE/[name of header file]". Note that the headers in this subdirectory are deleted each time the test is run, so do not store any project files in this directory.

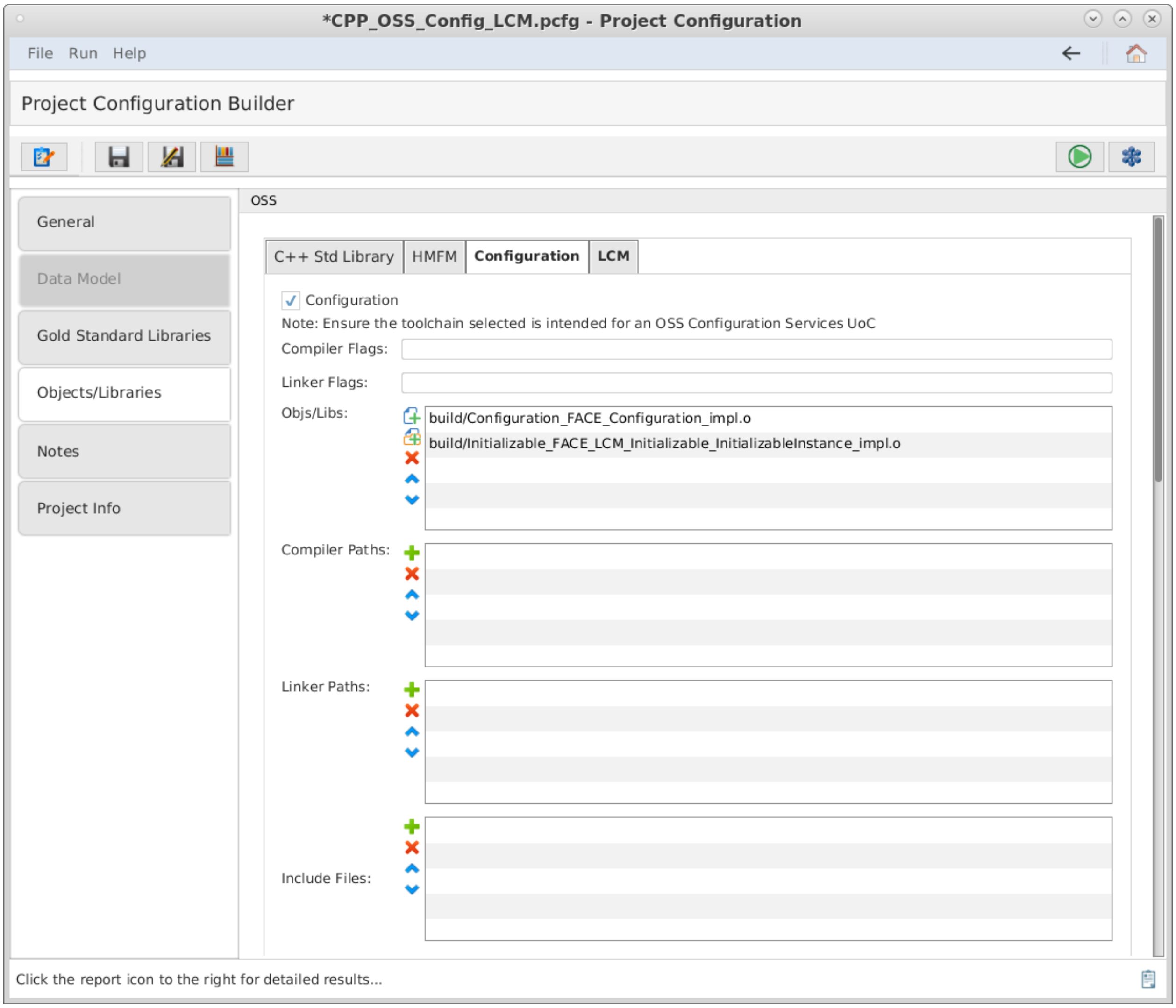
    The GSL libraries will be generated into the GSL Directory. You may wish to use the GSLs during development to check that your code builds against them, but there is no need to include them in your CTS project. The CTS will rebuild the appropriate GSLs when you run the test and link them as part of the test.

11. Enter the full pathnames of your project's object and/or library files. You may add each object file. You may also choose the directory where object files are located. All object files in the directory specified as well as object files in subdirectories will be chosen. Currently, library files must be chosen individually, not by directory. A combination of directories and object/library files may be specified. Note: Often system libraries are specified using compiler/linker flags instead of specifying libraries directly. Either method may be used.

12. **For providing FACE Configuration interface only, for C/C++/Ada only as of this version:** You must provide a source file that contains the implementation for each of the factory functions that creates an instance of each interface your UoC is providing. To determine which factory functions are necessary, complete the rest of this tutorial and click the "Generate GSLs" button (library icon) in the toolbar. Then, find the CTS_Factory_Functions.hpp header file in the generated subfolder

'include/FACE' within the Gold Standard Libraries folder you specified for the project. For C/C++, this file will contain the declarations of all expected factory functions that the CTS requires for testing your UoC. You must provide a source file that implements each of these functions. The implementation for each function must instantiate the corresponding concrete class and return a pointer to that object (the return type will be a pointer to the FACE base class). Returning a null pointer is not acceptable. This source file must be provided with your project and will be reviewed to ensure it instantiates your UoC's concrete class for that interface. For Ada, this will generate a spec file (.ads) cts_factory_functions.ads which has the procedures you must implement in the source file. The source file for Ada must be named "cts_factory_functions.adb" and implement each of these procedures, returning a concrete version of each type as an access type. In the text field labeled "Source file with factory functions for interfaces provided (C/C++/Ada only)", use the '...' button to the right to browse for the source file. **Any header files included by the source file must exist in one of the Include Paths specified above.** An example for a C++ project is shown below.

For Java, a source file named CTS_Factory_Functions.java will be generated in the factory/ subfolder (package subfolder) You must fill in the implementation of each function, add any imports, and add this to your project in the CTS in this file field. Note that this file will NOT be overwritten so if the interfaces the UoC uses have changed, you must delete your file in order to regenerate a new one with the new set of functions to implement.



☑ Configuration

Note: Ensure the toolchain selected is intended for an OSS Configuration Services UoC

Objs/Libs:
build/Configuration_FACE_Configuration_impl.o
build/Initializable_FACE_LCM_Initializable_InitializableInstance_impl.o

Click the report icon to the right for detailed results...

13. Select to verify that the Project Configuration File is valid.

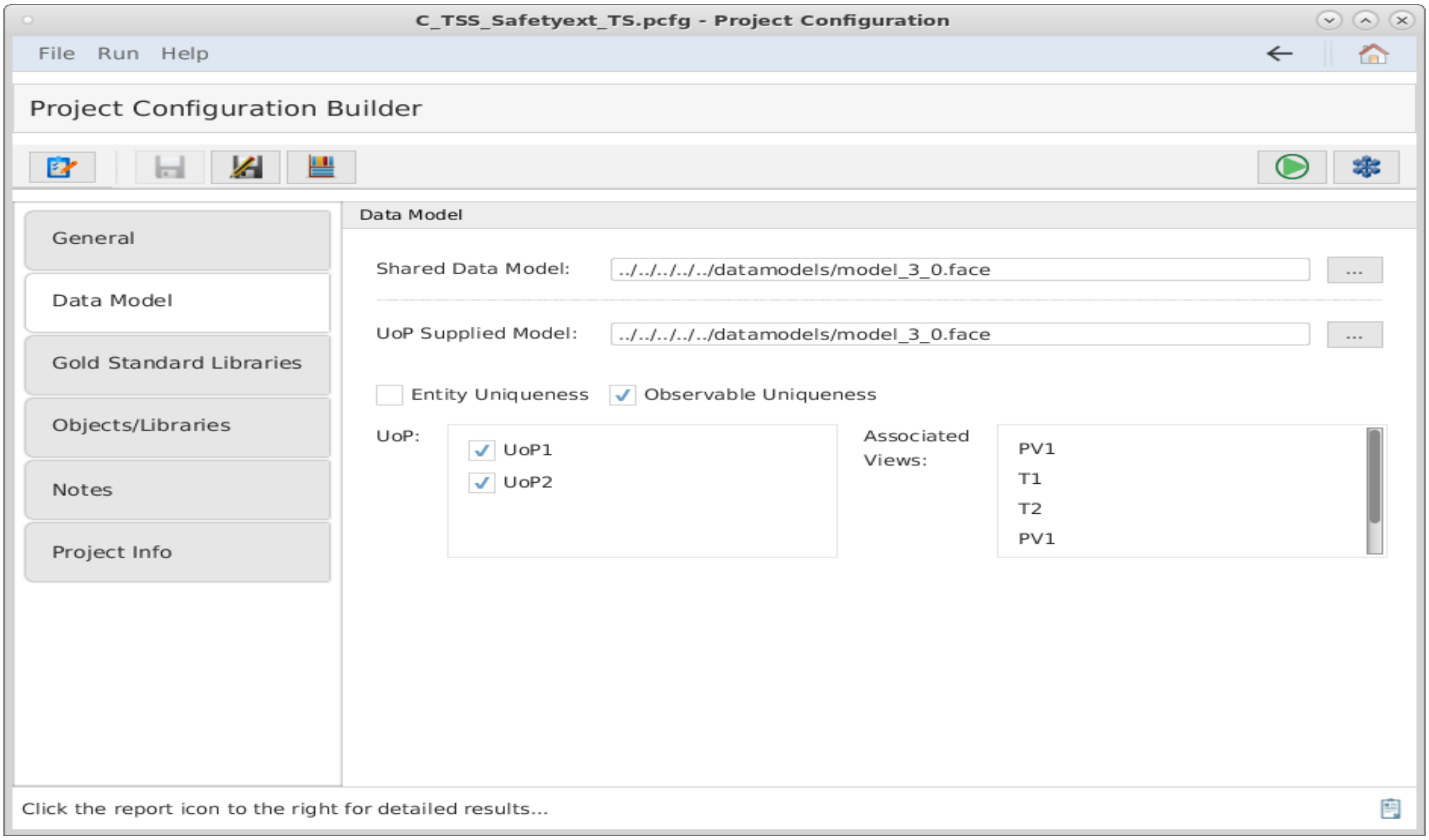
14. Click the button at the top of the screen to test the segment. (This may take a few minutes). The results will be written to a PDF file. The directory of the results file will be located in the directory of the project configuration file. This directory path will be listed in the "Output File Location" of the Conformance Test Results page.

15. The result will be pass or fail if the OS supplies the necessary calls based on the profile.

Testing a Data Model

What You Must Provide

- Your project's FACE data model file.



Shared Data Model: ../../../../../datamodels/model_3_0.face

UoP Supplied Model: ../../../../../datamodels/model_3_0.face

☐ Entity Uniqueness ☑ Observable Uniqueness

UoP:
☑ UoP1
☑ UoP2

Associated Views:
PV1
T1
T2
PV1

Click the report icon to the right for detailed results...

Procedure

1. Complete the procedure in the *Initialize Conformance Test* section above.
2. Assure that the PCS/PSS/TSS is selected in the General tab of the Project Configuration Builder.
3. Select the shared data model (SDM) file associated with the segment under test.
4. Optionally select the conditional Object Constraint Language (OCL) constraints governing USM and DSDM content.
    1. Select the Entity Uniqueness checkbox to define that the Entity is unique in a Conceptual Data Model.

    Note: An Entity is unique if the set of its Characteristics is different from other Entities' in terms of type, lowerBound, upperBound, and path (for Participants).

    1. Select the Observable Uniqueness checkbox to define that the Entity does not compose the same Observable more than once.

1. Select the UoP Supplied Model file associated with the segment under test.
2. The test suite will analyze the USM file, determining its validity and the Units of Portability found in the data model file.
3. You may see data types associated with a UoP by clicking on the properties button.

4. Select the Units of Portability to use with the segment under test.
5. Click the Test Data Model button.

   The results will be written to a PDF file. The directory of the PDF results will be located in the directory of the project configuration file. This directory path will be listed in the "Output File Location" of the Conformance Test Results page.

Considerations for Testing an Ada Segment

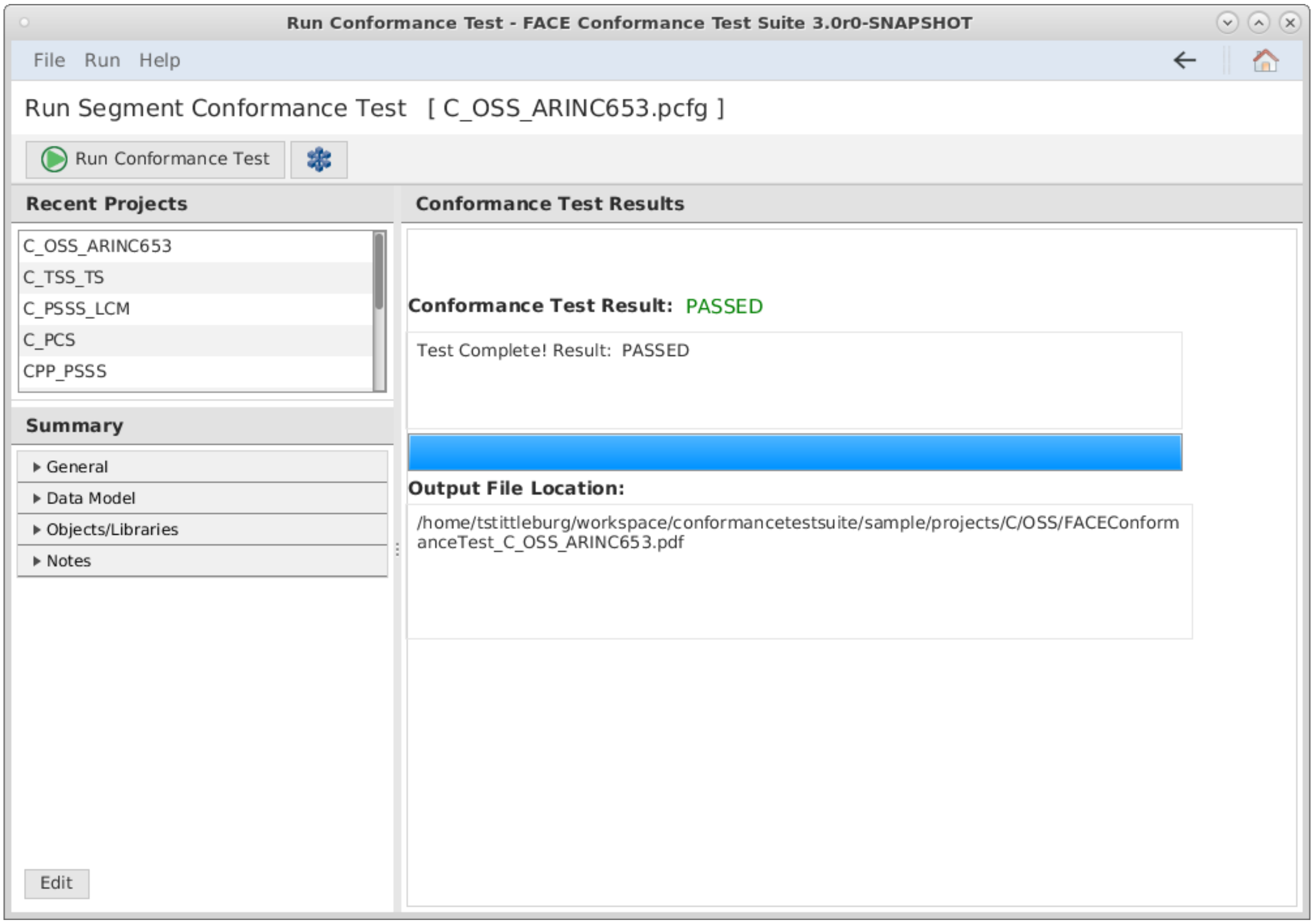
Testing an Ada segment requires a small variation in the testing procedures from C and C++. According to the standard, Ada Runtime Libraries are allowed, but if the Runtime Library is packaged with the UoP, it must only use standard POSIX calls allowed according to the profile/partition. If the Ada Runtime Libraries are part of the logical OSS, the use of the Ada Runtime Libraries is verified via Inspection. In order to perform the link test for a packaged Ada Runtime Library, you must include the Ada Runtime Library as part of your object/library files. Additionally, you must compile the correct Gold Standard POSIX library to include as part of your object library files. Since the test suite only supports compilation for one language at a time, you must build the POSIX libraries before proceeding with Ada testing. This can be done by changing your configuration from Ada to C, with the correct C compiler options, and generate the gold standard libraries as described below. Once the libraries have been built, change the configuration back to Ada, add the POSIX and Runtime libraries to your segment configuration and proceed with the test. The test suite does generate Ada gold standard HMFM and ARINC 653 libraries.

Considerations for Testing a Java Segment

Testing a Java segment is very different from testing procedures from other languages. Since Java is inspected directly instead of using a link test, there is not an option to generate gold libraries in Java. For each test, Java Class Paths are used instead of object/library files. Include paths are not used under Java tests. Under most systems, *javac* should be used as the compiler and *jar* should be used as the archiver. The object file extension should be set to *class* in the project's toolchain file.

Viewing Test Suite Results

Once the run Conformance test button is pressed, the test suite will conduct the conformance test and the results will be stored in PDF format. The file will be named FACEConformanceTest_Name_of_PCFG.pdf in the same directory as the pcfg file tested. All log files generated in the test will also be found in the log directory, although the same log files are found inside the PDF report. An example of a passable portable component is given in the figure below:



Run Segment Conformance Test [C_OSS_ARINC653.pcfg]

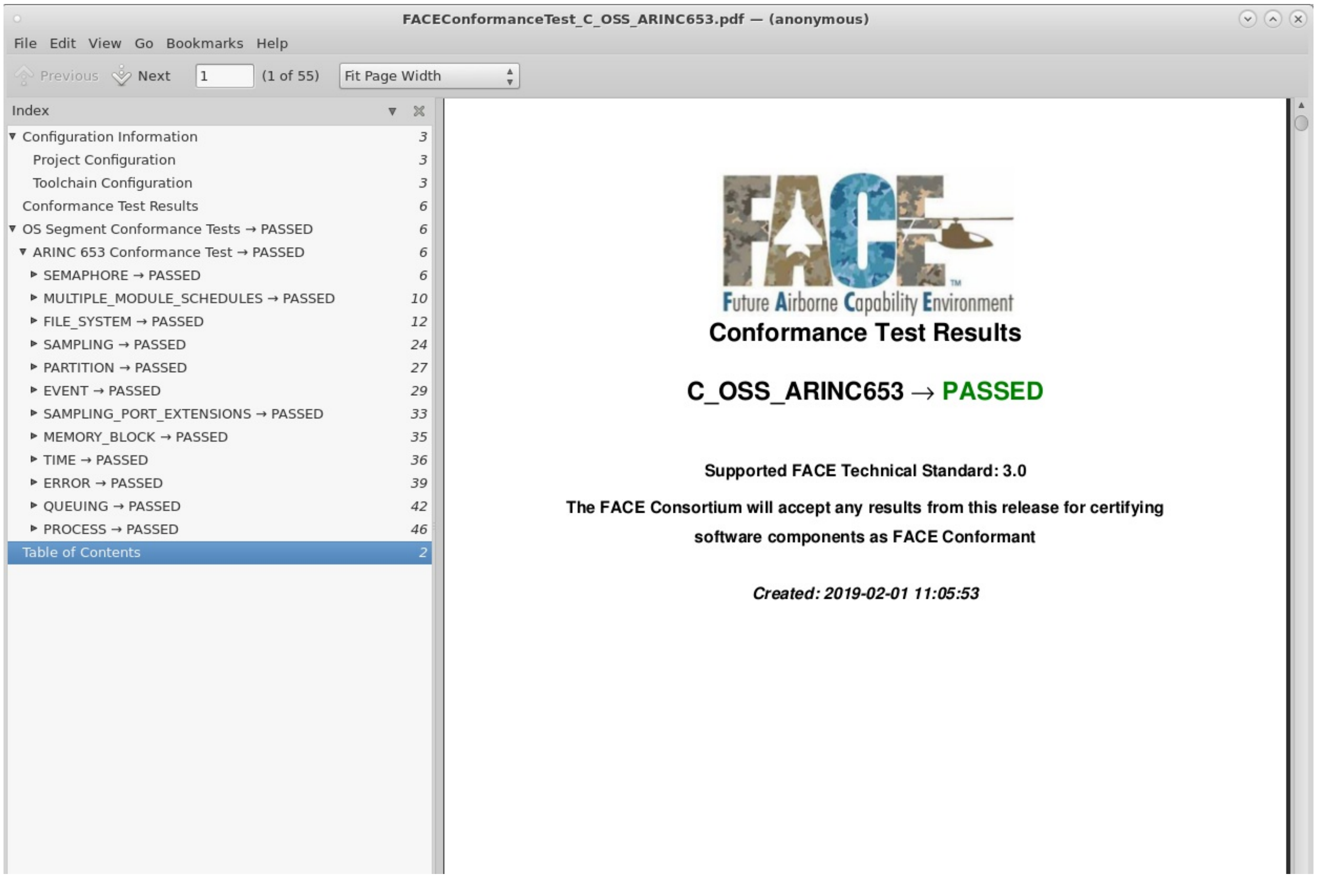
Conformance Test Result: PASSED

Test Complete! Result: PASSED

Output File Location:

/home/tstittleburg/workspace/conformancetestsuite/sample/projects/C/OSS/FACEConformanceTest_C_OSS_ARINC653.pdf

The output of the CTS is written to a PDF report. The path of the PDF report will be displayed in the Output File Location section.


FACEConformanceTest_C_OSS_ARINC653.pdf — (anonymous)
File Edit View Go Bookmarks Help
Previous
Next
1
(1 of 55)
Fit Page Width
Index

FACE™ Future Airborne Capability Environment

Conformance Test Results

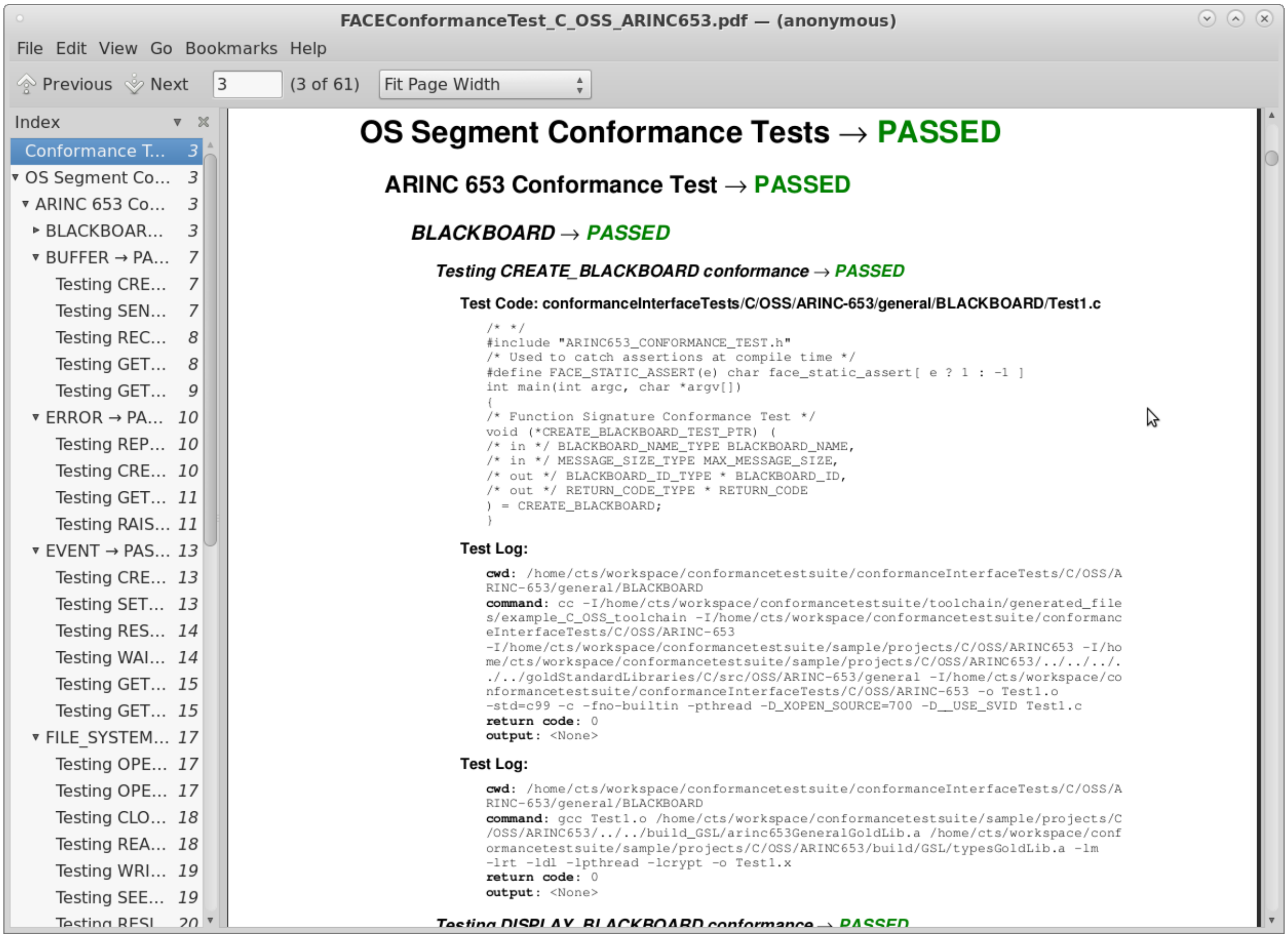
C_OSS_ARINC653 → PASSED

Supported FACE Technical Standard: 3.0

The FACE Consortium will accept any results from this release for certifying software components as FACE Conformant

Created: 2019-02-01 11:05:53

The PDF report will detail toolchain and project configuration information along with source code and/or log results associated with a test. Examples of the conformance test results can be seen below.



OS Segment Conformance Tests → PASSED

ARINC 653 Conformance Test → PASSED

BLACKBOARD → PASSED

Testing CREATE_BLACKBOARD conformance → PASSED

Test Code: conformanceInterfaceTests/C/OSS/ARINC-653/general/BLACKBOARD/Test1.c

```
/* */
#include "ARINC653_CONFORMANCE_TEST.h"
/* Used to catch assertions at compile time */
#define FACE_STATIC_ASSERT(e) char face_static_assert[ e ? 1 : -1 ]
int main(int argc, char *argv[])
{
/* Function Signature Conformance Test */
void (*CREATE_BLACKBOARD_TEST_PTR) (
/* in */ BLACKBOARD_NAME_TYPE BLACKBOARD_NAME,
/* in */ MESSAGE_SIZE_TYPE MAX_MESSAGE_SIZE,
/* out */ BLACKBOARD_ID_TYPE * BLACKBOARD_ID,
/* out */ RETURN_CODE_TYPE * RETURN_CODE
) = CREATE_BLACKBOARD;
}
```

Test Log:

```
cwd: /home/cts/workspace/conformancetestsuite/conformanceInterfaceTests/C/OSS/A
RINC-653/general/BLACKBOARD
command: cc -I/home/cts/workspace/conformancetestsuite/toolchain/generated_file
s/example_C_OSS_toolchain -I/home/cts/workspace/conformancetestsuite/conformanc
eInterfaceTests/C/OSS/ARINC-653
-I/home/cts/workspace/conformancetestsuite/sample/projects/C/OSS/ARINC653 -I/ho
me/cts/workspace/conformancetestsuite/sample/projects/C/OSS/ARINC653/../../../.
./../goldStandardLibraries/C/src/OSS/ARINC-653/general -I/home/cts/workspace/co
nformancetestsuite/conformanceInterfaceTests/C/OSS/ARINC-653 -o Test1.o
-std=c99 -c -fno-builtin -pthread -D_XOPEN_SOURCE=700 -D__USE_SVID Test1.c
return code: 0
output: <None>
```

Test Log:

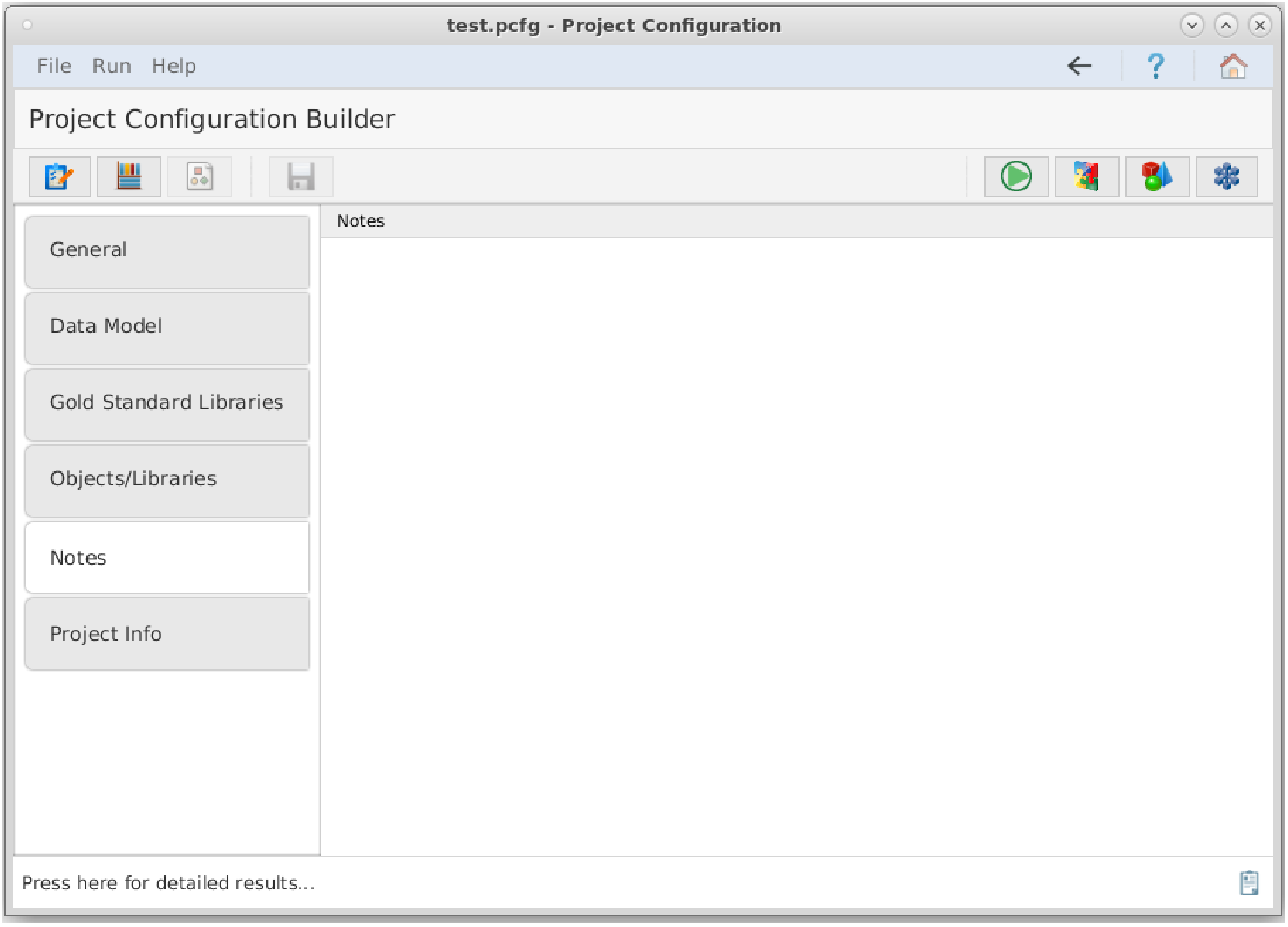
```
cwd: /home/cts/workspace/conformancetestsuite/conformanceInterfaceTests/C/OSS/A
RINC-653/general/BLACKBOARD
command: gcc Test1.o /home/cts/workspace/conformancetestsuite/sample/projects/C
/OSS/ARINC653/../../build_GSL/arinc653GeneralGoldLib.a /home/cts/workspace/conf
ormancetestsuite/sample/projects/C/OSS/ARINC653/build/GSL/typesGoldLib.a -lm
-lrt -ldl -lpthread -lcrypt -o Test1.x
return code: 0
output: <None>
```

Testing DISPLAY_BLACKBOARD conformance → PASSED

The source of the non-conformance for a failed conformance test can be determined by examining the test source code and resulting log files.

Including Test Notes with Configuration

Notes about partition tests or testing configurations may be added on the “Notes” tab. Any notes will be shown in the Test Configuration section of the conformance report.


test.pcfg - Project Configuration
File
Run
Help
Project Configuration Builder
General
Data Model
Gold Standard Libraries
Objects/Libraries
Notes
Project Info
Notes
Press here for detailed results...

Test Suite Command Line Options:

There are a number of options you can use when running the test suite start-up python script (conformance_test.py).

If the test suite is launched with a configuration file listed, the test suite will run without the GUI and save the results to the log directory listed in the configuration. The test suite will exit with a return code of 0 if the segment(s) under test is conformant. It will return 1 if the segment(s) fails conformance. This would be useful for automated testing of segments without user interaction.

Multiple configuration files can be passed to run by test suite, but it is important to have different log directories in each configuration file, otherwise the test results would be overwritten by subsequent tests.

The usage statement from the start-up script is shown below.

```
Usage: python conformance_test.py [options] [config_file1] [config_file2] ...
Options:
-h, --help                              Show this help message and exit.

-p PORT_VALUE, --port=PORT_VALUE        Used in coordination with the CTS_GUI. Port used to communicate with GUI
                                        supplied socket server.

-t, --time_stamp                        Generates a time stamp to be added to the report filename (assuring unique test
                                        run names).

-r REPORT_FILENAME, --                  Full path to the conformance test report PDF file. Default filename is
report_filename=REPORT_FILENAME         FACEConformanceTest_SEGMENT_PROJECT_NAME.pdf in the same
                                        directory as the segment project file.

-v, --version                           Verifies that a project configuration is valid for running conformance tests.
                                        Return code is 0 if valid, and 1 if invalid. Saves a
                                        log(PROJECT_CONFIG_NAME.ver_log) to the same directory as the
                                        configuration file, and sends the results to stdout.

-g --gold                               Build GSLs for given project.
-d --datamodel                          Test only data model.
```

Example Segments

The test suite has a sub-directory named “sample” that contains very simple examples of each segment in all four supported languages.

See the Installation section for instructions on building the sample projects.

Known Issues

If system headers are used instead of gold standard headers in compilation of segment software under test, a conforming segment may still fail the link test due to internal compiler issues. For example, using gcc and standard system headers, a conforming Portable Component has been known to fail due to an undefined reference to __stack_chk_fail. These issues can be resolved by adding allowable function calls to the CompilerSpecific gold standard library described above.

Note: The Configuration file structure has changed greatly from version 2.0 of the conformance test suite and cannot be ported into 3.0 conformance tests.

Acknowledgments

The test suite utilizes the following freely distributable software packages:

stringtemplate 3.1

http://www.stringtemplate.org/

Author: Benjamin Niemann

Protocol Buffers - Google's data interchange format

http://code.google.com/p/protobuf/

POSIX and ARINC interface testing is performed on functions only. Data types and constants are not tested comprehensively. A POSIX or ARINC conformance test should be used to fully test those aspects.