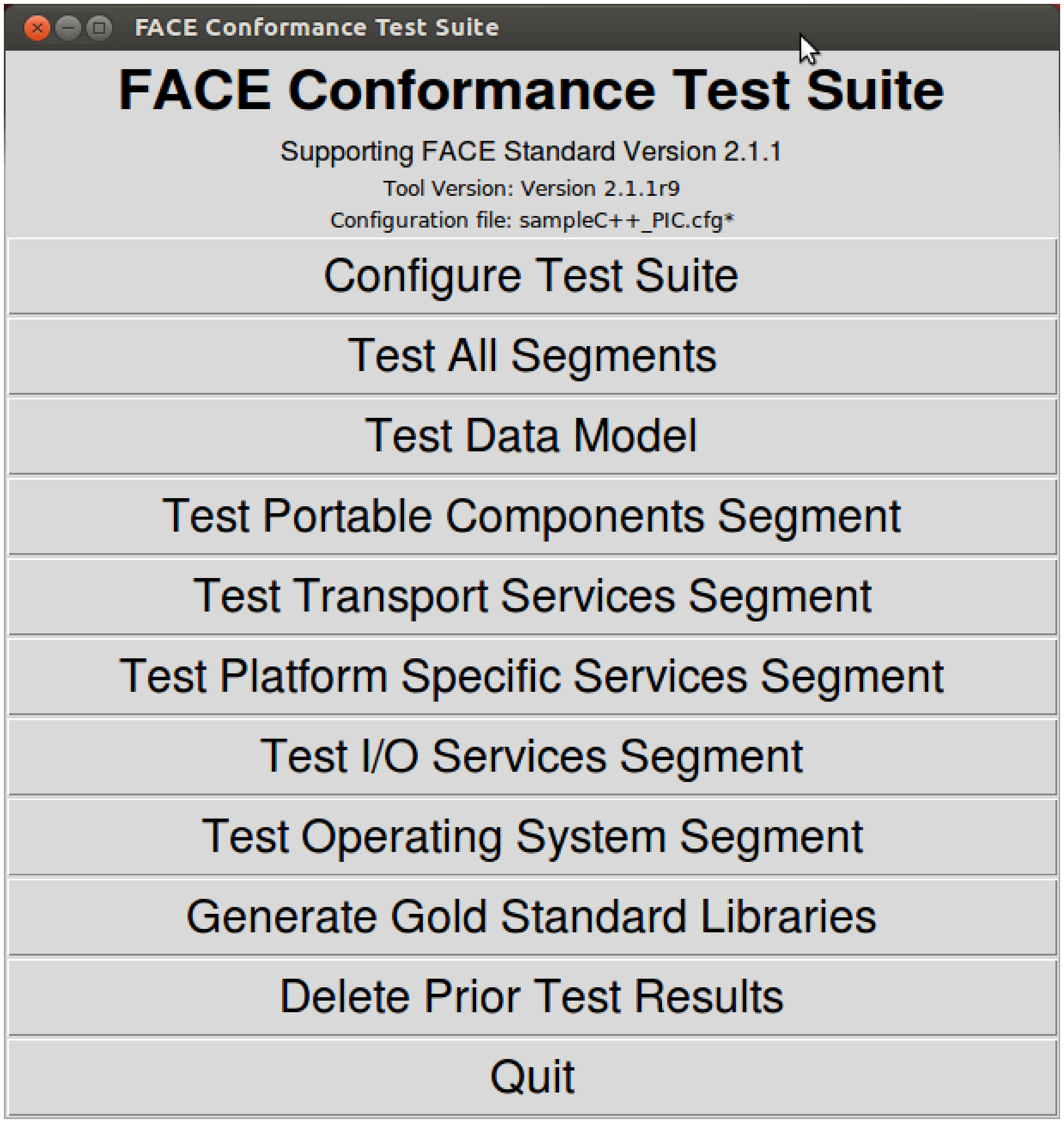
Candidate Conformance Test Suite Manual For Testing Interface And Application Code Against The FACE™ Standard 2.1.1


FACE Conformance Test Suite
FACE Conformance Test Suite
Configure Test Suite
Test All Segments
Test Data Model
Test Portable Components Segment
Test Transport Services Segment
Test Platform Specific Services Segment
Test I/O Services Segment
Test Operating System Segment
Generate Gold Standard Libraries
Delete Prior Test Results
Quit

Supporting FACE Standard Version 2.1.1
Tool Version: Version 2.1.1r9
Configuration file: sampleC++_PIC.cfg*

Acronyms Used

TOG	The Open Group
FACE	Future Airborne Capability Environment
PCS	Portable Components Segment
PSS	Platform Specific Services
TSS	Transport Services Segment
IOS	Input/Output (I/O) Segment
OS	Operating System
UoP	Unit of Portability
USM	UoP Supplied Model
SDM	Shared Data Model
FGSL	FACE Gold Standard Libraries

Table of Contents

Introduction

The Conformance Test Suite tests the following interfaces and applications for FACE Standard Conformance:

1. Portable Components Segment (PCS) applications
2. Platform Specific Services (PSS) Segment applications
3. Transport Services Segment (TSS) interfaces
4. I/O Services (IOS) Segment interfaces
5. OS (POSIX and ARINC 653) interfaces

Testing is performed under the OS Partitions POSIX and ARINC 653 and the OS Profiles General Purpose, Safety Base, Safety Extended, and Security. Testing procedures for each interface and application type are listed in the chapters below.

Installation And Configuration

System Requirements

Before installation check the system requirements below to ensure the test suite will run on your designated machine. The conformance test suite runs in either a Windows or Linux based computer system. The software has been tested using Microsoft Windows 7 and Microsoft Windows 10. Linux systems tested include Ubuntu 12.x and RHEL/CentOS 7.

Common system requirements

- Python 2.6+ (not 3.x, in Windows, be sure that the python executable is in your PATH)
- TkInter
- Java 1.8 JRE (be sure that Java is in your PATH)
- Browser (Firefox, Chromium, Chrome, Opera, Konquerer, Epiphany, or Internet Explorer)
- Associated Compiler/Linker/Archiver used for the software under conformance test.
- Recommended 4GB+ RAM.

Optional software

The test suite can use the included pymake (https://developer.mozilla.org/en-US/docs/pymake) in it's build process, however any GNU compatible make can be used. The native make on Linux systems is considerably faster. MinGW and Cygwin have been used on Microsoft Windows systems.

Installation

Select an installation location and decompress the test suite. To launch the test suite, execute the runConformanceTest script (runConformanceTest.bat for Windows, runConformanceTest.sh for Linux). The script checks for proper installation of Python, Java, and TkInter before launching the Test Suite program.

To conduct Java testing, an additional download and installation is required. Download the Java add-on package for your platform and decompress the package into the top level of the conformance tool such that CONFORMANCE_TOOL_ROOT/Java_Conformance/JavaConformance(.exe) exists on your system.

If you're performing Java testing on Windows, you will additionally need to download and install this Microsoft runtime: https://www.microsoft.com/en-us/download/details.aspx?id=26999 Failure to do so will result in a message similar to the following: “The program can't start because MSVCP100.dll is missing from your computer”.

Configuration

There is compiler specific information that will be needed to conduct conformance tests . This information is stored in the compilerSpecific subdirectory.

NULL:

The NULL type must be configured according to your compiler's manual. This is done by edit the following two files:

- compilerSpecific/C/systemLibraryDefinitions/CONFORMANCE_TEST_FACE_NULL.h
- compilerSpecific/C++/systemLibraryDefinitions/CONFORMANCE_TEST_FACE_NULL.h

Consult your compiler's manual to create the macro definition. The macro name you must define is FACE_NULL. Do not define NULL itself which will be handled by the test suite based on your definition of FACE_NULL.

Exact Types:

For C and C++ testing, the exact size types must be configured according to your compiler's manual. This is done by editing the following two files:

- compilerSpecific/C/systemLibraryDefinitions/CONFORMANCE_TEST_FACE_EXACT_TYPES.h
- compilerSpecific/C++/systemLibraryDefinitions/CONFORMANCE_TEST_FACE_EXACT_TYPES.h

Consult your compiler's manual to create a typedef mapping between its intrinsic types and the exact types needed for testing. The exact types needed are described in the table below.

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()

Most exact types will be covered by signed char, signed short, signed int, signed long, unsigned char, unsigned short, unsigned int, and unsigned long. However, those types may not cover the full spectrum of types necessary for testing. Your compiler may or may not provide other compiler specific intrinsic types. Those types are typically but not always prefixed with underscores. The following C and C++ programs can be used to assist in determining the types.

```
#include <stddef.h>
#include <stdio.h>

int main()
{
  printf("signed char    is %2zd bits\n", sizeof(signed char   ) * 8);
  printf("signed short   is %2zd bits\n", sizeof(signed short  ) * 8);
  printf("signed int     is %2zd bits\n", sizeof(signed int    ) * 8);
  printf("signed long    is %2zd bits\n", sizeof(signed long   ) * 8);
  printf("unsigned char  is %2zd bits\n", sizeof(unsigned char ) * 8);
  printf("unsigned short is %2zd bits\n", sizeof(unsigned short) * 8);
  printf("unsigned int   is %2zd bits\n", sizeof(unsigned int  ) * 8);
  printf("unsigned long  is %2zd bits\n", sizeof(unsigned long ) * 8);
  printf("size_t         is %2zd bits\n", sizeof(size_t        ) * 8);

  return 0;
}

#include <cstddef>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << "signed char    is " << setw(2) << sizeof(signed char   ) * 8 << " bits" << endl;
  cout << "signed short   is " << setw(2) << sizeof(signed short  ) * 8 << " bits" << endl;
  cout << "signed int     is " << setw(2) << sizeof(signed int    ) * 8 << " bits" << endl;
  cout << "signed long    is " << setw(2) << sizeof(signed long   ) * 8 << " bits" << endl;
  cout << "unsigned char  is " << setw(2) << sizeof(unsigned char ) * 8 << " bits" << endl;
  cout << "unsigned short is " << setw(2) << sizeof(unsigned short) * 8 << " bits" << endl;
  cout << "unsigned int   is " << setw(2) << sizeof(unsigned int  ) * 8 << " bits" << endl;
  cout << "unsigned long  is " << setw(2) << sizeof(unsigned long ) * 8 << " bits" << endl;
  cout << "size_t         is " << setw(2) << sizeof(size_t        ) * 8 << " bits" << endl;

  return 0;
}
```

OpenGL:

OpenGL libraries used by FACE have compiler specific details per their specifications. Macros, types, and function decoration are among the things defined in these headers. Many implementations will choose to use the Khronos provided headers. Other implementations may define their own. Find these header files from your implementation of choice. Copy them into the compiler specific directory as specified below. If those headers have other dependencies such as POSIX, be sure to select the use of POSIX in your test configuration. If those headers have other non-FACE dependencies those must be included in the compiler specific directory, and a Verification Authority will decide on their validity.

- compilerSpecific/C/systemLibraryDefinitions/EGL/eglplatform.h
- compilerSpecific/C/systemLibraryDefinitions/GLES2/gl2platform.h
- compilerSpecific/C/systemLibraryDefinitions/KHR/khrplatform.h
- compilerSpecific/C++/systemLibraryDefinitions/EGL/eglplatform.h
- compilerSpecific/C++/systemLibraryDefinitions/GLES2/gl2platform.h
- compilerSpecific/C++/systemLibraryDefinitions/KHR/khrplatform.h

If those headers have other dependencies such as POSIX, be sure to select the use of POSIX in your test configuration. If those headers have other non-FACE dependencies those must be included in the compiler specific directory, and a Verification Authority will decide on their validity.

Allowed Definitions:

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 in the CompilerSpecific/LANGUAGE/allowedDefinitions directory. You only need to add a function stub, since these conformance test objects are never actually executed. Compiler specific methods must be reported to the VA (and are included in the Test Suite results). For all segments other than OSS, the compiler specific files system library definition files (NULL, EXACT TYPES, OpenGL) should not change for a given tool chain. The allowed definitions may vary. However, **when testing an OSS, the allowed definitions file should be empty, since you are testing the operating system libraries, and all necessary functions should be defined.**

The CTS expects Portable Components and Platform-Specific Services Segment (PCS, PSSS) components to have main functions in their provided object files. Without a main function the conformance test results are invalid. If a PCS or PSSS under test do not contain a main function, a main function that fully exercises the component will need to be added to the conformance test. This is most easily done by adding this to the "CompilerSpecific" source file and adding the required include path to the compiler flags in the configuration. Alternatively, a separate object containing the main function can be added to the object files that are undergoing conformance testing.

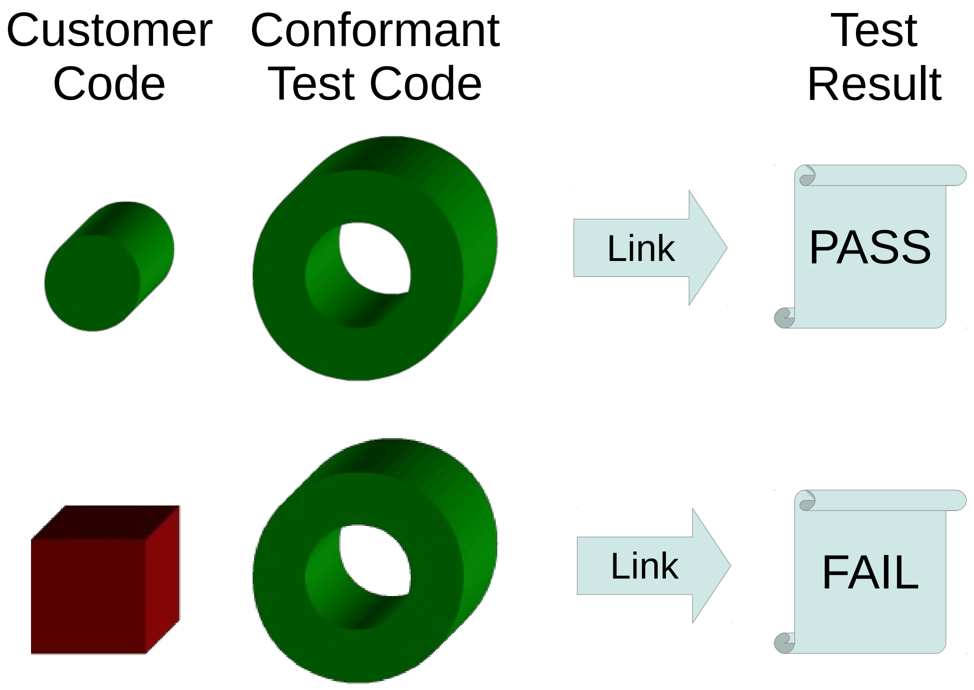
Note: All source files in the CompilerSpecific/LANGUAGE/allowedDefinitions directory are used to make the compilerSpecificGoldLib.a by the CTS during testing. The CTS automatically cleans the object files in these directories between test runs. The user may choose to explicitly clean up these object files by executing "make clean" in the CompilerSpecific/LANGUAGE directory.

Note: The sample directory contains examples of compiler specific files that may be valid for your configuration.

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 application. The test applications utilize all possible function calls, data types, and constants that should be provided by the customer interface[1]. If the link passes, the customer code is conformant. If the link fails, the customer code is not conformant. Errors are included in the test output.

The link test only determines conformance with respect to function signature. The link test neither proves nor disproves correctness of functionality or correctness of function usage.



For Java, a lookup table of classes and method signatures is built from a whitelist of Java APIs and FACE interfaces from the standard. The lookup table also incorporates all classes supplied

1 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.

by the customer, preferring the class and method signatures derived by the standard where duplication may exist. The table serves as the “gold standard” of allowable and/or expected classes and methods. All supplier class files passed into the CTS are parsed and evaluated against the lookup table to ensure the classes and methods used match what is in the lookup table. Any classes and methods not found in the lookup table but used in supplier code is flagged as error and included in the test output.

Introduction to Methodology

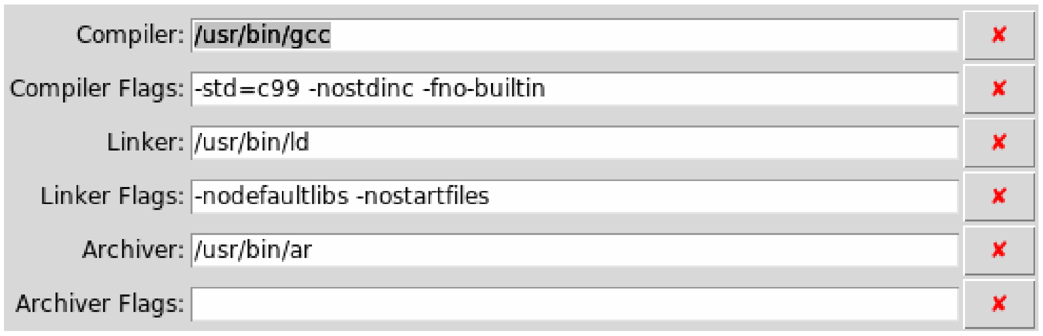
Two methods of performing the link test exist. One uses the target linker. The other uses the host linker. The target linker is the linker used to produce an executable targeting the embedded system. The host linker is the linker used to produce an executable targeting the PC where the conformance test suite 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.

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. For all but OSS tests, 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.

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.



Compiler:	/usr/bin/gcc
Compiler Flags:	-std=c99 -nostdinc -fno-builtin
Linker:	/usr/bin/ld
Linker Flags:	-nodefaultlibs -nostartfiles
Archiver:	/usr/bin/ar
Archiver Flags:	

Example flags for a non-OSS C segment test

Commonly Used Compiler Flags:

The table below provides the minimum set of equivalences you must provide.

Flag Purpose		***GNU Tools Example***
Language standard selection		
C	ISO C 1999	-std=c99
C++	ISO C++ 2003	-std=c++03 (or c++0x on some compilers)
Ada	ISO Ada 1995	-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
C++ has language features which may need to be disabled based on the FACE Profile or program coding conventions. The below compiler arguments are specific to GCC.		
Do not use or allow exceptions.		-fno-exceptions
Do not register static object destructors to be called from the 'atexit()' method. 'atexit()' is not in the Safety Base Profile and systems are not expected to exit.		-fno-use-cxa-atexit
Do not generate code that allows trapping instructions to throw exceptions. An example of this is floating point errors.		-fno-non-call-exceptions
Do not use or allow Run-Time Type Information.		-fno-rtti
Do not generate exceptions for NULL being returned by 'operator new'.		-fcheck-new

Note: It has been determined that using compiler optimization flags (-O, etc) has caused invalid conformance test results. Please do not use compiler optimization flags.

Additional Methodology Information

When building your project, alter your compiler flags to include the conformance tool's goldStandardLibraries directory for IOS, TSS, and OS 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. The compiler specific allowed definitions should also be blank.

C++ Testing Methodology

For profiles other than the general profile, if the standard template library is desired to be used for a non-OS segment, it must be provided as part of the unit of conformance.

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. CTS supports Oracle SE 8’s JVM class file format.

PCS, PSSS, TSS and IOSS UoC class files are queried for their dependencies on 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. Additionally, native methods are flagged as warnings for further inspection.

UoCs responsible for providing FACE interfaces (e.g., OSS, TSS, and IOSS UoCs) will have their class files queried for the list of classes, methods, and fields they are expected to provide as defined by the standard. Any omissions or incorrect definitions are reported as errors.

Note that all segment class files are checked for their usage of Java SE and EE (general profile only) version 8 gold definitions. Third Party class definitions that provide a standardized runtime environment and has been approved by FACE Consortium can be identified using an entry box called Approved Framework on the PCS, PSS, TSS, and IOS configuration tabs of the CTS GUI. The Approved Framework class definitions are added to the white list during evaluation of segment class files.

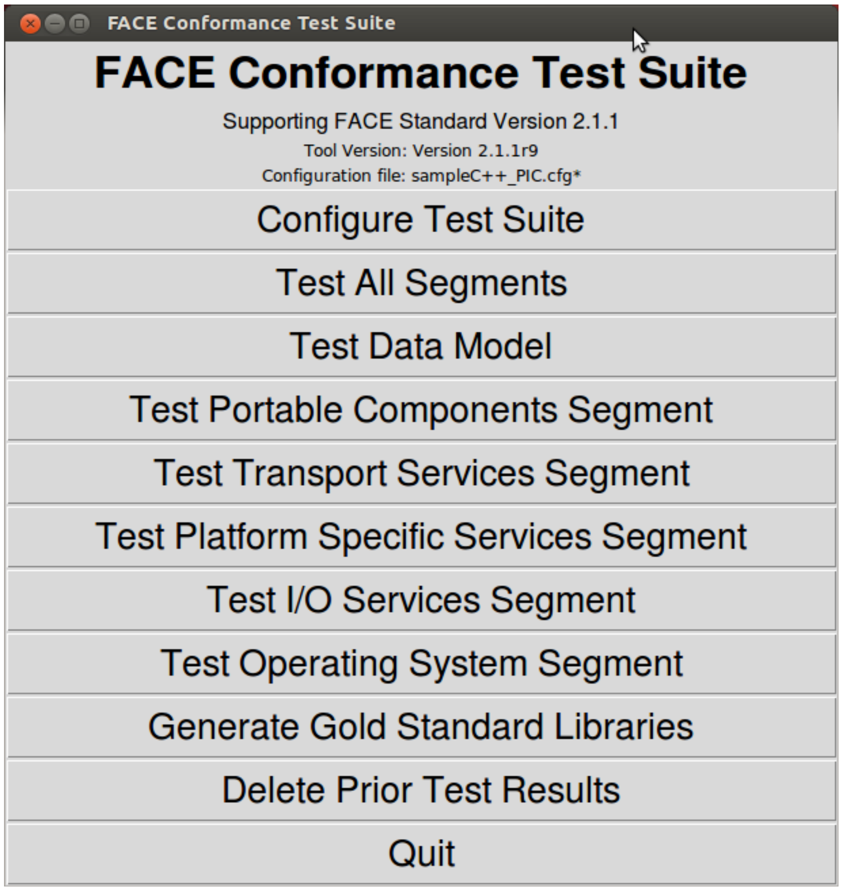
Initializing the Conformance Test Suite

What You Must Provide

- The programming language used in the segment(s) under test
- The OS Profile to utilize
- The OS Partition to utilize
- A directory for the log files and other artifacts
- A browser to display the conformance test results
- Build tool (compiler/linker/archiver) information (see Methodology above)

Procedure

1. Make sure you have performed the exact types configuration as described above, if applicable.
2. Start the conformance test suite by running the runConformanceTest script in the main test suite directory from the command line (runConformanceTest.bat for Windows, runConformanceTest.sh for Linux). This will launch the conformance main menu as shown below:



FACE Conformance Test Suite

Supporting FACE Standard Version 2.1.1

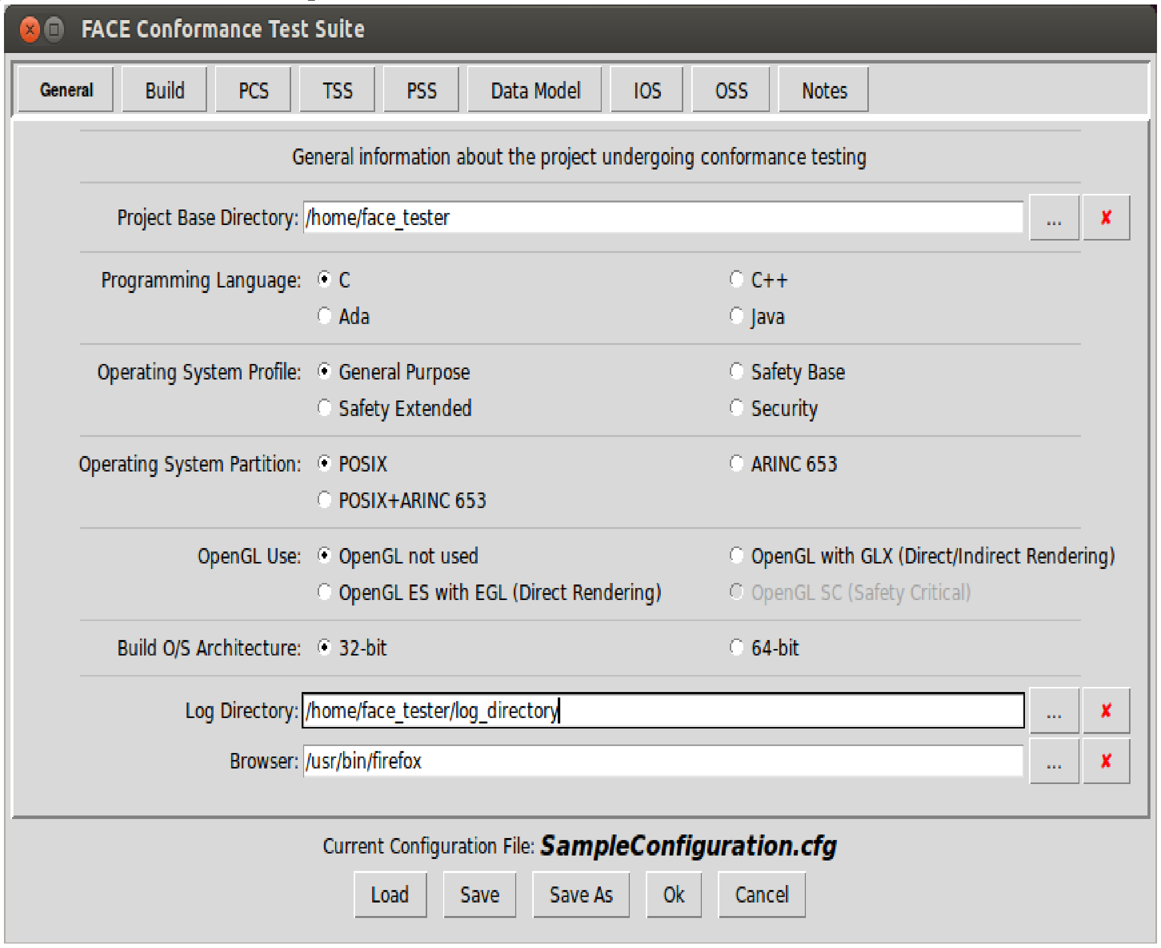
Tool Version: Version 2.1.1r9

Configuration file: sampleC++_PIC.cfg*

FACE Conformance Test Suite
Configure Test Suite
Test All Segments
Test Data Model
Test Portable Components Segment
Test Transport Services Segment
Test Platform Specific Services Segment
Test I/O Services Segment
Test Operating System Segment
Generate Gold Standard Libraries
Delete Prior Test Results
Quit

3. Click the “Configure Test Suite” button to launch the configure dialog box.

4. Select the "General" tab at the top of the configuration dialog box to display the generation information options as shown below:



FACE Conformance Test Suite

General | Build | PCS | TSS | PSS | Data Model | IOS | OSS | Notes

General information about the project undergoing conformance testing

Project Base Directory: /home/face_tester

Programming Language: [✓ C] C++ Ada Java

Operating System Profile: [✓ General Purpose] Safety Base Safety Extended Security

Operating System Partition: [✓ POSIX] ARINC 653 POSIX+ARINC 653

OpenGL Use: [✓ OpenGL not used] OpenGL with GLX (Direct/Indirect Rendering) OpenGL ES with EGL (Direct Rendering) OpenGL SC (Safety Critical)

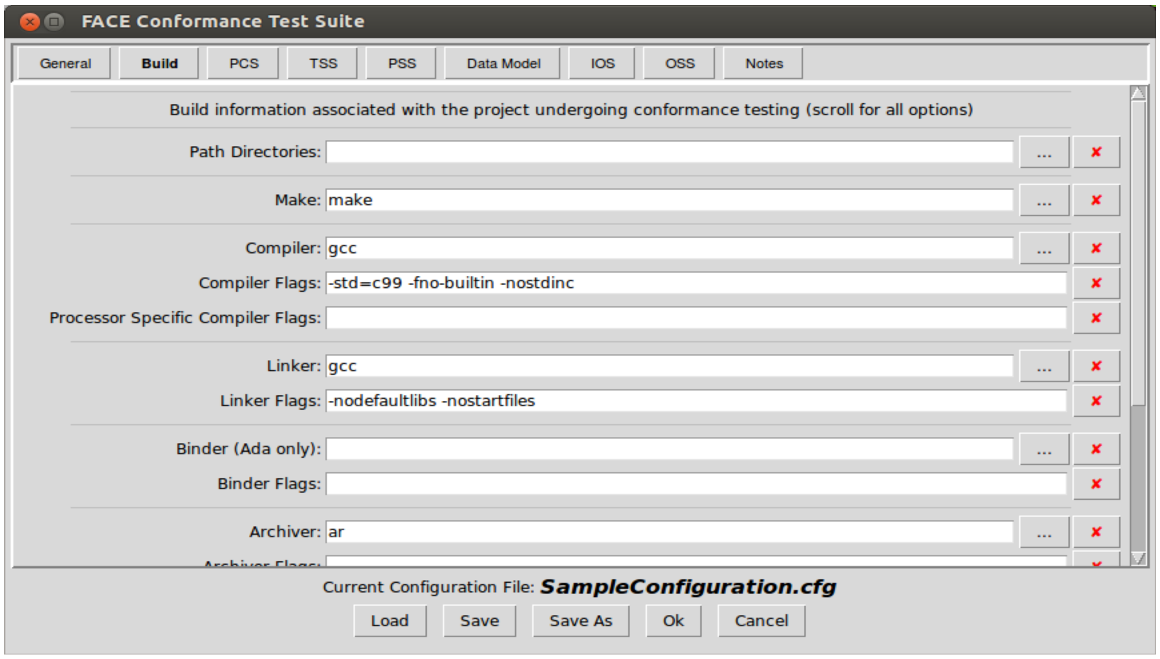
Build O/S Architecture: [✓ 32-bit] 64-bit

Log Directory: /home/face_tester/log_directory

Browser: /usr/bin/firefox

Current Configuration File: ***SampleConfiguration.cfg***

5. (Optional) Select the project directory where the segment(s) interface software under test is located. This will set the default directory for later steps in the configuration procedure and will allow the tester to only modify this entry if the root directory of the software changes. If the Project Base Directory is changed once other project configuration information is entered. The test suite will automatically change to the new directory and will notify the tester if there are inconsistencies with the new directory.
6. Select the programming language used by the segment(s) interface under test.
7. Select the operating system profile to be used by the segment(s) under test.
8. Select the operating system partition to be used by the segment(s) under test.
9. Select the OpenGL configuration to be used by the segment(s) under test.
10. Select the build OS architecture.
11. Choose the log directory to store the log files and all artifacts generated by the test.
12. Choose the location of the browser to use to display the test suite results. The test suite will pick a default browser if none is specified. Since this may take a few seconds on some systems, it is encouraged to set up the basic configuration on the general and build tabs and save as the default configuration. See "Setting Default Configuration" on the following page for more information.
13. Select the "Build" tab to the top of the configuration dialog box to display the general build options as shown below:



Build information associated with the project undergoing conformance testing (scroll for all options)

Path Directories:
Make: make
Compiler: gcc
Compiler Flags: -std=c99 -fno-builtin -nostdinc
Processor Specific Compiler Flags:
Linker: gcc
Linker Flags: -nodefaultlibs -nostartfiles
Binder (Ada only):
Binder Flags:
Archiver: ar

Current Configuration File: ***SampleConfiguration.cfg***

Example build system configuration - Your development environment may require different values

14. You will need to scroll down to see all build options.
15. Choose any directories required to be in the path in order to compile, link and archive code using your desired build tools. If using MinGW on Windows, the MinGW\bin and the MinGW\msys\1.0\bin directories should be included. No test suite related directories are required under Linux.
16. Choose the GNU-make compatible executable for the test suite to use. Type "pymake" if using the included pymake software for the make option. The test suite will automatically determine the appropriate path and command. Otherwise, the make executable can be specified by the full path name, or if the path is included in the Path Directories entry box, the file name is sufficient.
17. Choose the build options for compiler, linker, and archiver according to the Theory of Operation section above. A field for processor specific compiles flags has been provided. Please do not use any optimization flags for your build setup, since these can cause errors in conformance testing. 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++).*
18. Choose the extension used for object files by the compiler.
19. Choose the extension used for executable files by the compiler. (Leave blank for no extension.)
20. The Data Model analysis is written in Java. The test suite launches a JVM to run the data model analysis during testing. For larger data models, the default heap size may not be large enough. It is recommended to specify your heap size with the -Xmx java command line option. (For a 2GB heap, you would list *-Xmx2048m* in the Java command line options line.

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.

Setting Default Configuration

To make a given configuration the default when the test suite is started, save it to the initialConfigurationWINDOWS.cfg or initialConfigurationLINUX.cfg in the face_conformance_app subdirectory. It is suggested that the general and build tabs are configured correctly for the build environment and then saved as the initial configuration.

Providing your configuration file to the verification authority would assure they have your correct configuration for their conformance testing. By default these files are saved in the configFiles sub-directory.

Compiler Specific Functionality

You will need to modify the compiler specific files as described in the configuration section above.

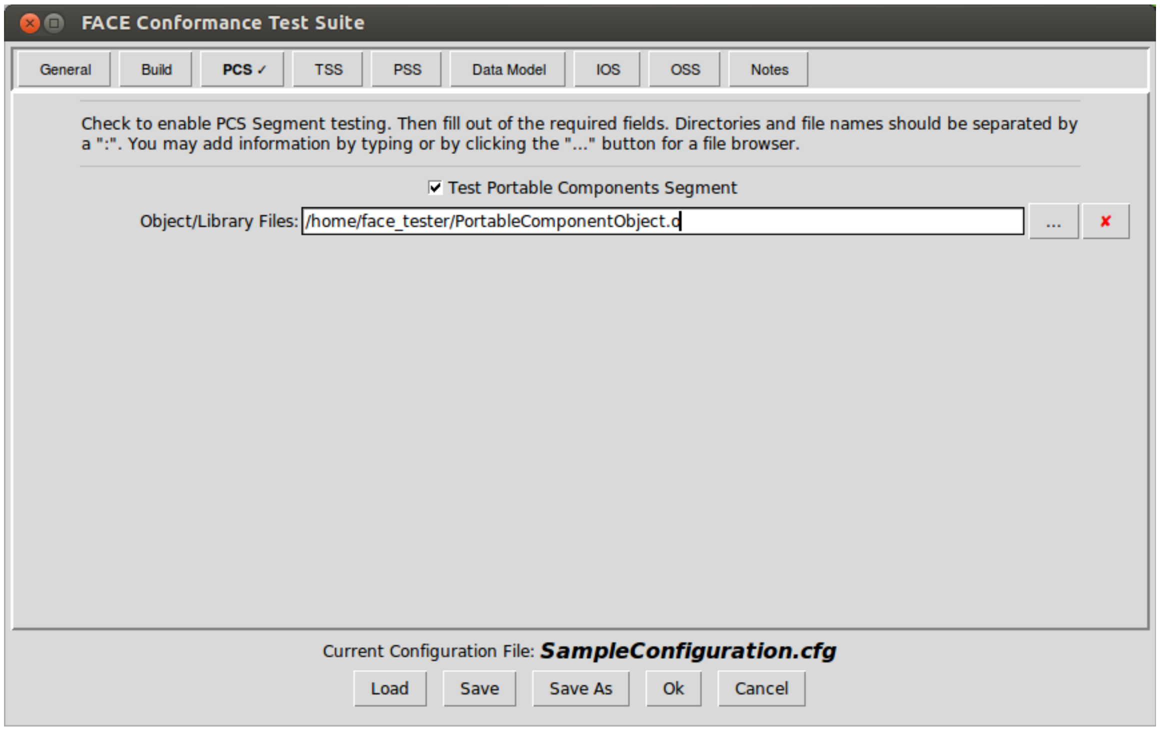
Testing a Portable Components Segment (PCS) Application

What You Must Provide

- Your project's object files
- Your project's data model

Procedure

1. Compile (but do not link) your project. (see Additional Methodology Information above)
2. Complete the procedure in the *Initialize Conformance Test* section above.
3. Select the PCS tab at the top of the configuration dialog box to display the portable components information options as shown below:



Check to enable PCS Segment testing. Then fill out of the required fields. Directories and file names should be separated by a ":". You may add information by typing or by clicking the "..." button for a file browser.

✓ Test Portable Components Segment

Object/Library Files: /home/face_tester/PortableComponentObject.o

Current Configuration File: ***SampleConfiguration.cfg***

4. Enter the full pathnames your project's object and/or library files. You may use the "…" button to add your project files graphically. You may either specify each object and library file, or you may chose 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 and object/library files may be specified.
5. Check the Portable Components Segment check box. Notice the check mark on the PCS tab.
6. Complete the procedure in the *Data Model* section below.
7. Click the Save As button to store this configuration for future use. You may click the Load button to reuse this configuration in the future.
8. Click the "Ok" button to accept this configuration.

9. Click the “Test Portable Components Segment” button on the main menu to test the segment. (This may take a few minutes.) The results will be displayed in a web browser when testing is complete. Artifacts (including the report shown in the web browser) will be placed in the log directory you specified above.

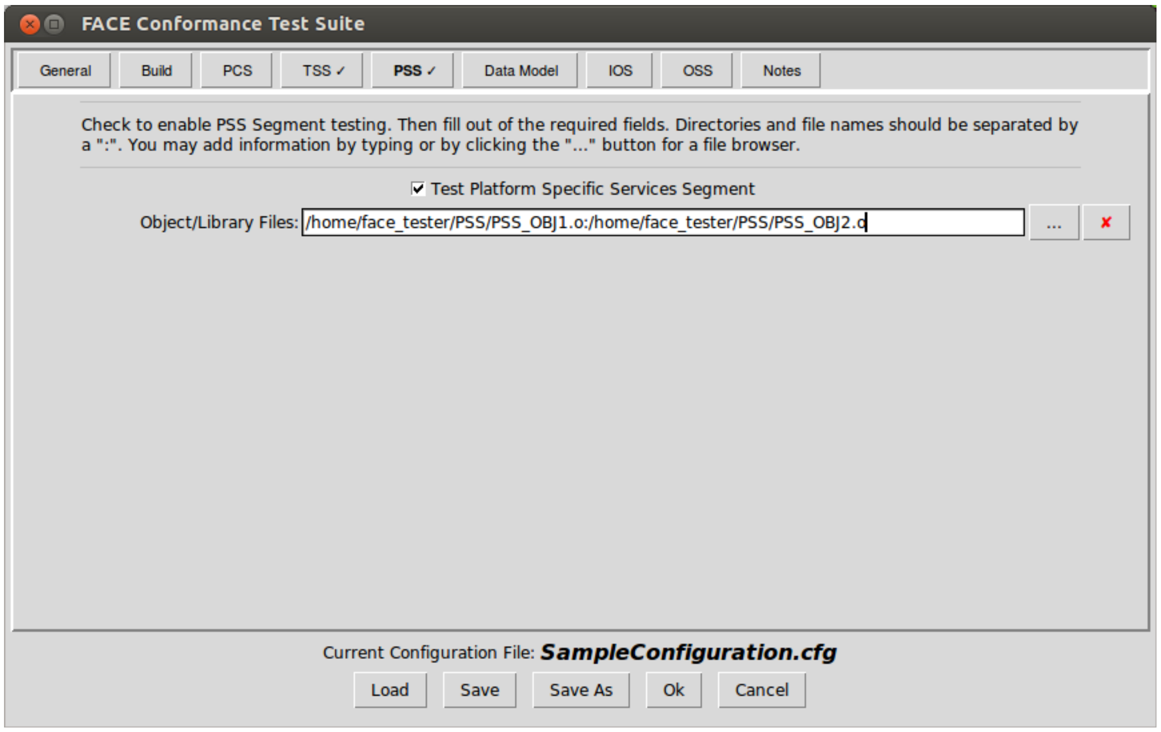
Testing a Platform Specific Services (PSS) Segment

What You Must Provide

- Your project's object files
- Your project's data model

Procedure

1. Compile (but do not link) your project. (see Additional Methodology Information above)
2. Complete the procedure in the *Initialize Conformance Test* section above.
3. Select the PSS tab at the top of the configuration dialog box to display the platform specific information options as shown below:



Check to enable PSS Segment testing. Then fill out of the required fields. Directories and file names should be separated by a ":". You may add information by typing or by clicking the "..." button for a file browser.

☑ Test Platform Specific Services Segment

Object/Library Files: /home/face_tester/PSS/PSS_OBJ1.o:/home/face_tester/PSS/PSS_OBJ2.o

Current Configuration File: ***SampleConfiguration.cfg***

4. Enter the full pathnames your project's object and/or library files. You may use the “…” button to add your project files graphically. You may either specify each object and library file, or you may chose 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 and object/library files may be specified.
5. Check the Platform Specific Services Segment check box. Notice the check mark on the PSS tab.

6. Complete the procedure in the *Data Model* section below
7. Click the Save As button to store this configuration for future use. You may click the Load button to reuse this configuration in the future.
8. Click the “Ok” button to accept this configuration.
9. Click the “Test Portable Platform Specific Services Segment” button on the main menu to test the segment. (This may take a few minutes.) The results will be displayed in a web browser when testing is complete. Artifacts (including the report shown in the web browser) will be placed in the log directory you specified above.

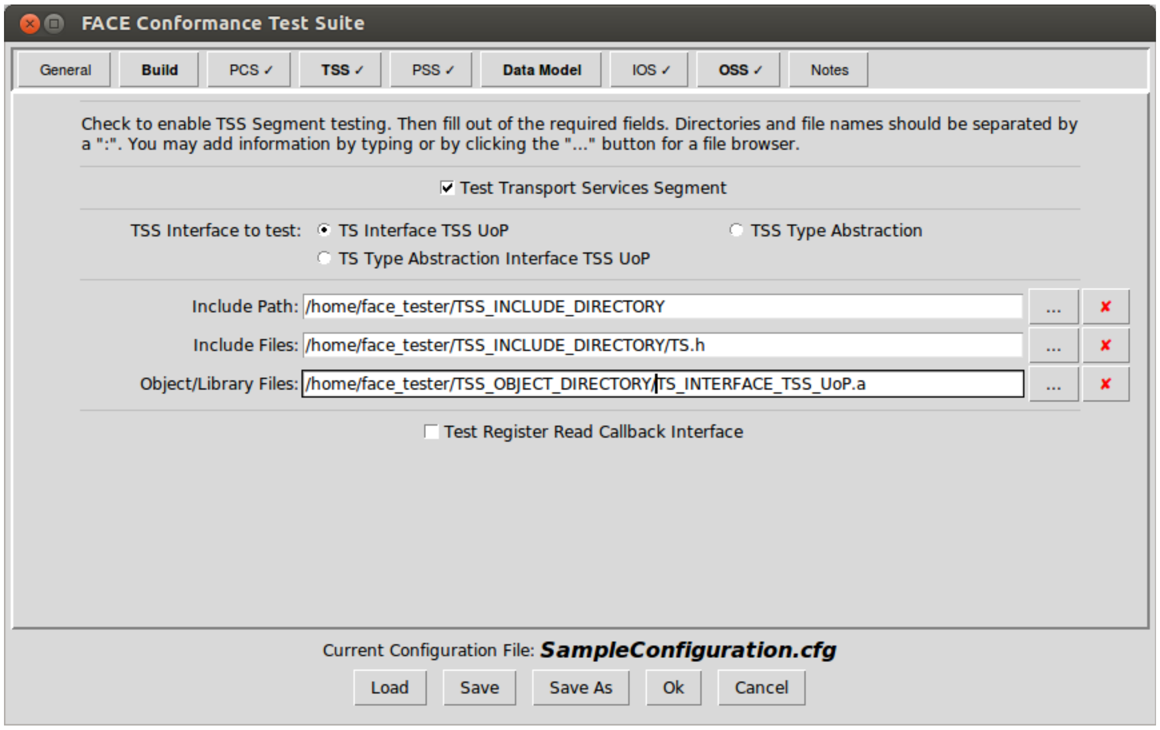
Testing a Transport Services Segment (TSS)

What You Must Provide

- Your project's include path
- Your project's header files
- Your project's object files
- Your project's data model

Procedure

1. Compile (but do not link) your project. (see Additional Methodology Information above)
2. Complete the procedure in the *Initialize Conformance Test* section above.
3. Select the TSS tab at the top of the configuration dialog box to display the transport services information options as shown below:



Check to enable TSS Segment testing. Then fill out of the required fields. Directories and file names should be separated by a ":". You may add information by typing or by clicking the "..." button for a file browser.

[✓] Test Transport Services Segment

TSS Interface to test: [✓ TS Interface TSS UoP] TSS Type Abstraction TS Type Abstraction Interface TSS UoP

Include Path: /home/face_tester/TSS_INCLUDE_DIRECTORY

Include Files: /home/face_tester/TSS_INCLUDE_DIRECTORY/TS.h

Object/Library Files: /home/face_tester/TSS_OBJECT_DIRECTORY/TS_INTERFACE_TSS_UoP.a

[ ] Test Register Read Callback Interface

Current Configuration File: ***SampleConfiguration.cfg***

4. Check the Transport Services Segment check box. Notice the check mark on the TSS tab.
5. Select the type of TSS interface you wish to test.
    - The TS Interface TSS UoP will test that your object files contain all TS interfaces required by the standard for the given data model and use only OSS calls that are allowed for a TSS.
    - The TSS Type Abstraction will test that your object files contain all TS interfaces required by the standard for the specified data model and use only interfaces to a TS Type Abstraction Interface UoP and OSS calls that are allowed for a TSS.
    - The TS Type Abstraction Interface TSS UoP will test that your object files contain all TS interfaces required by the standard and use only OSS calls that are allowed for a TSS.
6. Enter any directories that should be in the include path for the TSS interface. You may use the "…" button to add your project files graphically.
7. Enter the full pathnames of your project's TSS include/spec files. You may use the "…" button to add your project files graphically.
8. Enter the full pathnames of your project's object and/or library files. You may use the "…" button to add your project files graphically. You may either specify each object and library file, or you may chose 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 and object/library files may be specified.
9. If using register read callback interface, click the associated check box.
10. Complete the procedure in the *Data Model* section below.
11. Click the Save As button to store this configuration for future use. You may click the Load button to reuse this configuration in the future.
12. Click the "Ok" button to accept this configuration.
13. Click the "Test Transport Services Segment" button on the main menu to test the segment. (This may take a few minutes.) The results will be displayed in a web browser when testing is complete. Artifacts (including the report shown in the web browser) will be placed in the log directory you specified above.

Testing a Data Model

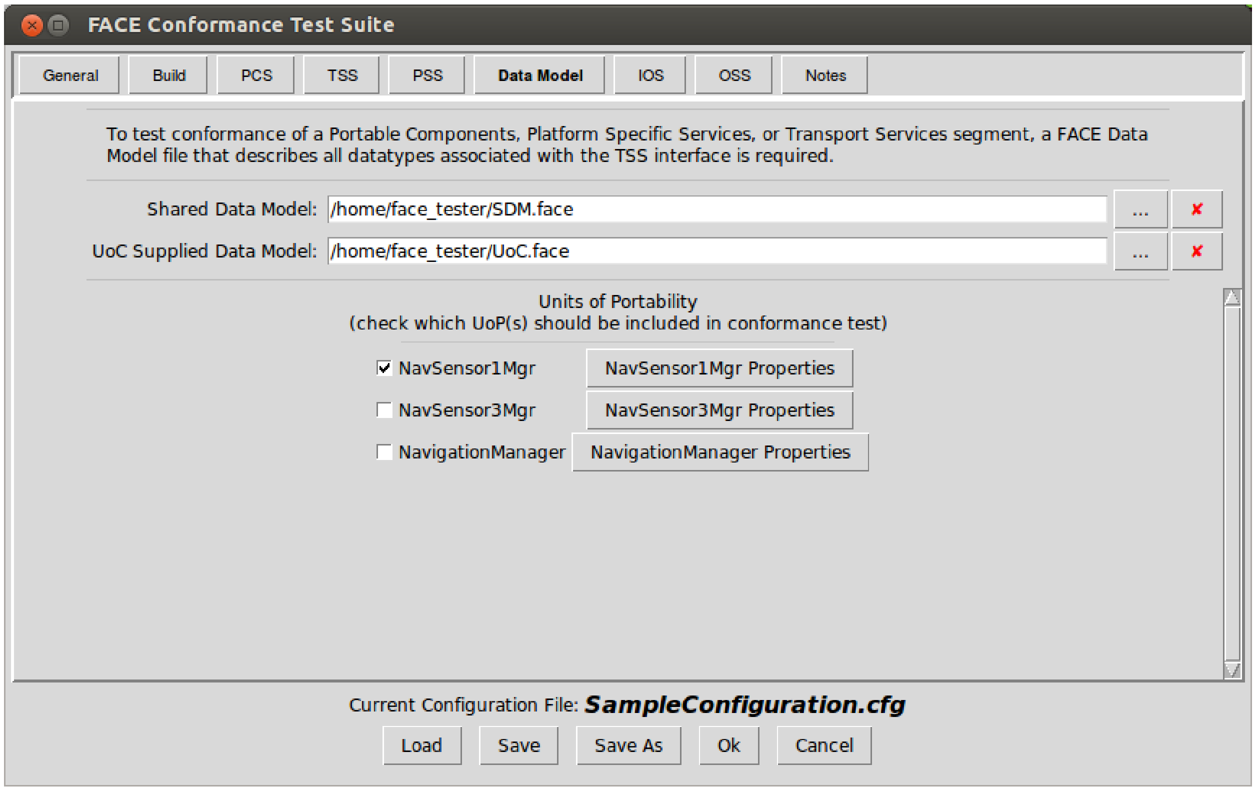
What You Must Provide

- Your project's FACE data model file.

Procedure

1. Complete the procedure in the *Initialize Conformance Test* section above.
2. Assure that at least one of the check boxes for either the PCS, TSS, or PSS is selected on their respective configuration tab dialogs.

3. Select the "Data Model" tab to display the data type information options as shown below:



To test conformance of a Portable Components, Platform Specific Services, or Transport Services segment, a FACE Data Model file that describes all datatypes associated with the TSS interface is required.

Shared Data Model: /home/face_tester/SDM.face

UoC Supplied Data Model: /home/face_tester/UoC.face

Units of Portability
(check which UoP(s) should be included in conformance test)

- [✓] NavSensor1Mgr — NavSensor1Mgr Properties
- [ ] NavSensor3Mgr — NavSensor3Mgr Properties
- [ ] NavigationManager — NavigationManager Properties

Current Configuration File: ***SampleConfiguration.cfg***

4. Select the shared data model file associated with the segment under test.
5. Select the UoC supplied data model file associated with the segment under test.
6. The test suite will analyze the UoC Data Model file, determining its validity and the Units of Portability found in the data model file.
7. You may see data types associated with a UoP by clicking on the properties button.
8. Select the Units of Portability to use with the segment under test.

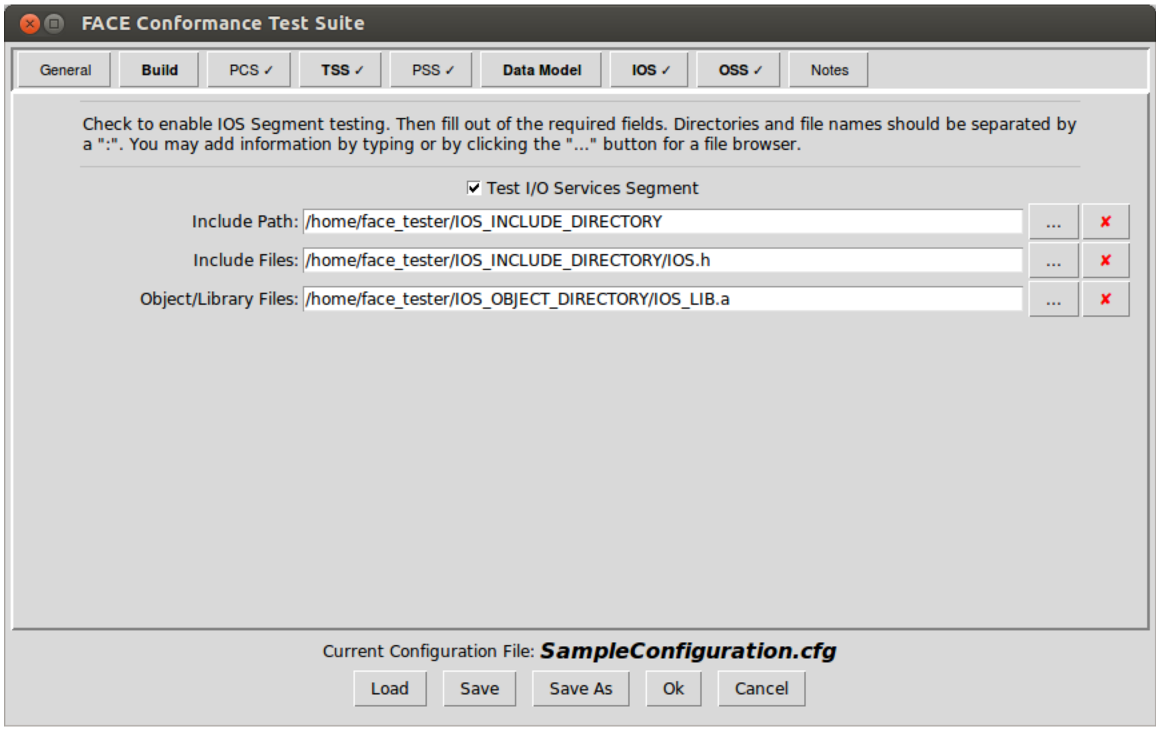
Testing an I/O Services (IOS) Segment

What You Must Provide

- Your project's include path
- Your project's header files
- Your project's object files

Procedure

1. Compile (but do not link) your project. (see Additional Methodology Information above)
2. Complete the procedure in the *Initialize Conformance Test* section above.
3. Select the IOS tab at the top of the configuration dialog box to display the platform specific information options as shown below:



Check to enable IOS Segment testing. Then fill out of the required fields. Directories and file names should be separated by a ":". You may add information by typing or by clicking the "..." button for a file browser.

☑ Test I/O Services Segment

Include Path: /home/face_tester/IOS_INCLUDE_DIRECTORY

Include Files: /home/face_tester/IOS_INCLUDE_DIRECTORY/IOS.h

Object/Library Files: /home/face_tester/IOS_OBJECT_DIRECTORY/IOS_LIB.a

Current Configuration File: ***SampleConfiguration.cfg***

4. Check the I/O Services Segment check box. Notice the check mark on the IOS tab.
5. Enter any directories that should be in the include path for the IOS interface. You may use the "…" button to add your project files graphically.
6. Enter the full pathnames of your project's IOS include/spec files. You may use the "…" button to add your project files graphically.
7. Enter the full pathnames your project's object and/or library files. You may use the "…" button to add your project files graphically. You may either specify each object and library file, or you may chose 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 and object/library files may be specified.
8. Click the Save As button to store this configuration for future use. You may click the Load button to reuse this configuration in the future.
9. Click the "Ok" button to accept this configuration.
10. Click the "Test I/O Services Segment" button on the main menu to test the segment. (This may take a few minutes.) The results will be displayed in a web browser when testing is complete. Artifacts (including the report shown in the web browser) will be placed in the log directory you specified above.

Testing an Operating System (OSS) Segment

What You Must Provide

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

Procedure

1. Complete the procedure in the *Initialize Conformance Test* section above.
2. Select the OSS tab at the top of the configuration dialog box to display the platform specific information options as shown below:
3. Click the Test Operation System Segment check box. 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 table below for more details.

Language/Profile	ARINC 653	C Std Library	C++ Std Libary	HMFM	Java	Khronus Group EGL 1.2	OpenGL ES 2.0	OpenGL SC 1.0.1	POSIX
C/GP	X	X		X		X	X		X
C/SB, C/SE	X	X		X				X	X
C/S	X	X		X					X
C++/All			X	X					
Ada/All	X			X					
Java/GP					X				

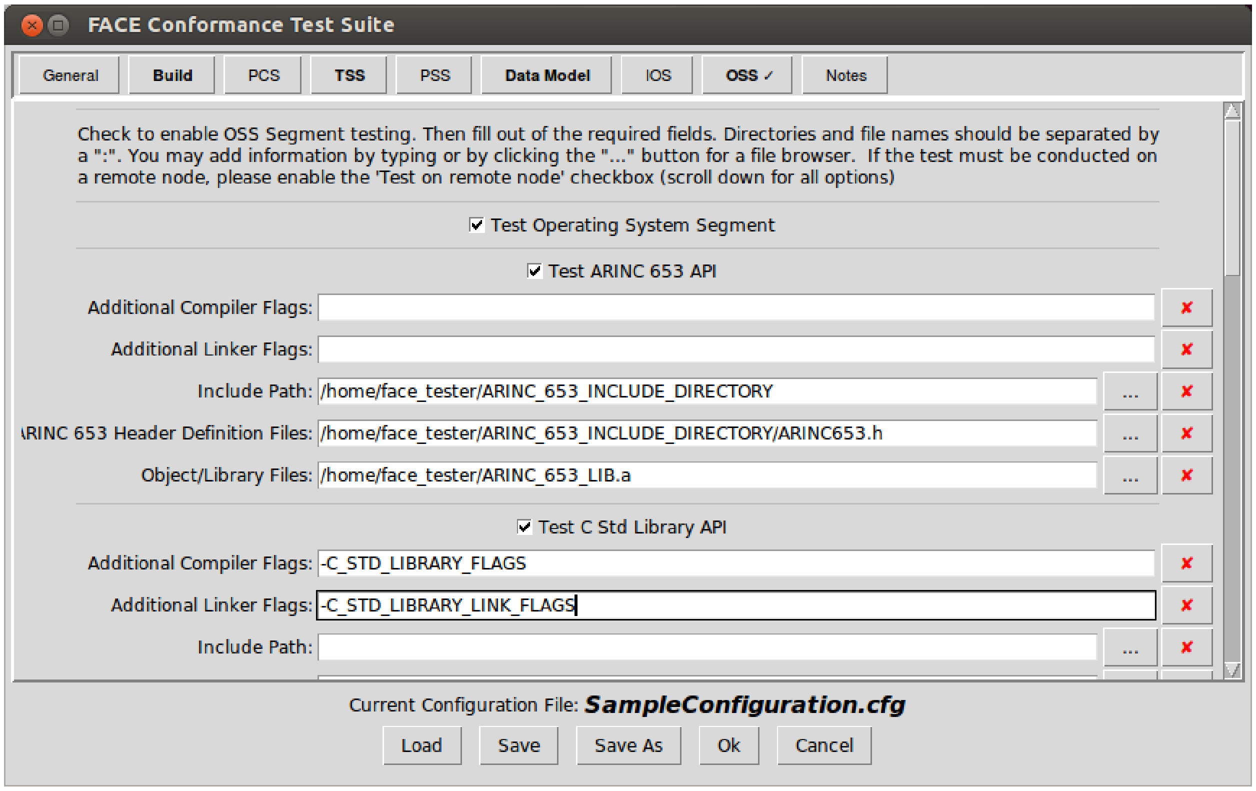
Note: GP=General Purpose, SB=Safety Base, SE=Safety Extended, S=Security, All=All profiles

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. You may use the "…" button to add your project files graphically.
8. If the ARINC 653 API is under test, enter the full pathnames to the header definition files associated with the interface. You may use the "…" button to add your project files graphically.
9. Enter the full pathnames your project's object and/or library files. You may use the "…" button to add your project files graphically. You may either specify each object and library file, or you may chose 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 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.*
10. Click the Save As button to store this configuration for future use. You may click the Load button to reuse this configuration in the future.
11. Click the "Ok" button to accept this configuration.
12. Click the "Test Operating System Segment" button on the main menu to test the segment. (This may take a few minutes.) The results will be displayed in a web browser when

testing is complete. Artifacts (including the report shown in the web browser) will be placed in the log directory you specified above.

13. The result will be pass or fail if the operating system supplies the necessary calls based on the profile.



Check to enable OSS Segment testing. Then fill out of the required fields. Directories and file names should be separated by a ":". You may add information by typing or by clicking the "..." button for a file browser. If the test must be conducted on a remote node, please enable the 'Test on remote node' checkbox (scroll down for all options)

[✓] Test Operating System Segment

[✓] Test ARINC 653 API

Additional Compiler Flags:	
Additional Linker Flags:	
Include Path:	/home/face_tester/ARINC_653_INCLUDE_DIRECTORY
\RINC 653 Header Definition Files:	/home/face_tester/ARINC_653_INCLUDE_DIRECTORY/ARINC653.h
Object/Library Files:	/home/face_tester/ARINC_653_LIB.a

[✓] Test C Std Library API

Additional Compiler Flags:	-C_STD_LIBRARY_FLAGS
Additional Linker Flags:	-C_STD_LIBRARY_LINK_FLAGS
Include Path:	

Current Configuration File: ***SampleConfiguration.cfg***

Considerations for Testing a C++ Segment

When testing a C++ PC, PSS, IOS or TS UoC in a safety or security profile that uses C++ Standard Template Library API calls, a FACE conformant standard template library implementation must be supplied with the UoC.

Considerations for using C and C++ together

It is very common for C++ programs to utilize C code. The FGSL allow this as well. The libstdc++ profiles have the C99 headers that are specified in the C++2003 standard. If POSIX headers or all C99 headers are required, please include the respective directory in the include path when compiling your objects to undergo conformance testing. The POSIX directory will need to come before the libstdc++ directory in your include path. The FGSL libstdc++ libraries do not contain C functions, but the allowed clib or POSIX library are automatically included under a C++ link test depending on the profile and partition.

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 for 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. Either pick the directory(s) or pick jar files. Note that directories can contain .class files or .jar files. Do not pick .class files individually. Include paths are not used under Java tests. Under most systems, *javac* should be used as the compiler (1.8 version) and *jar* should be used as the archiver, and the object file extension should be set to *class* under the build tab.

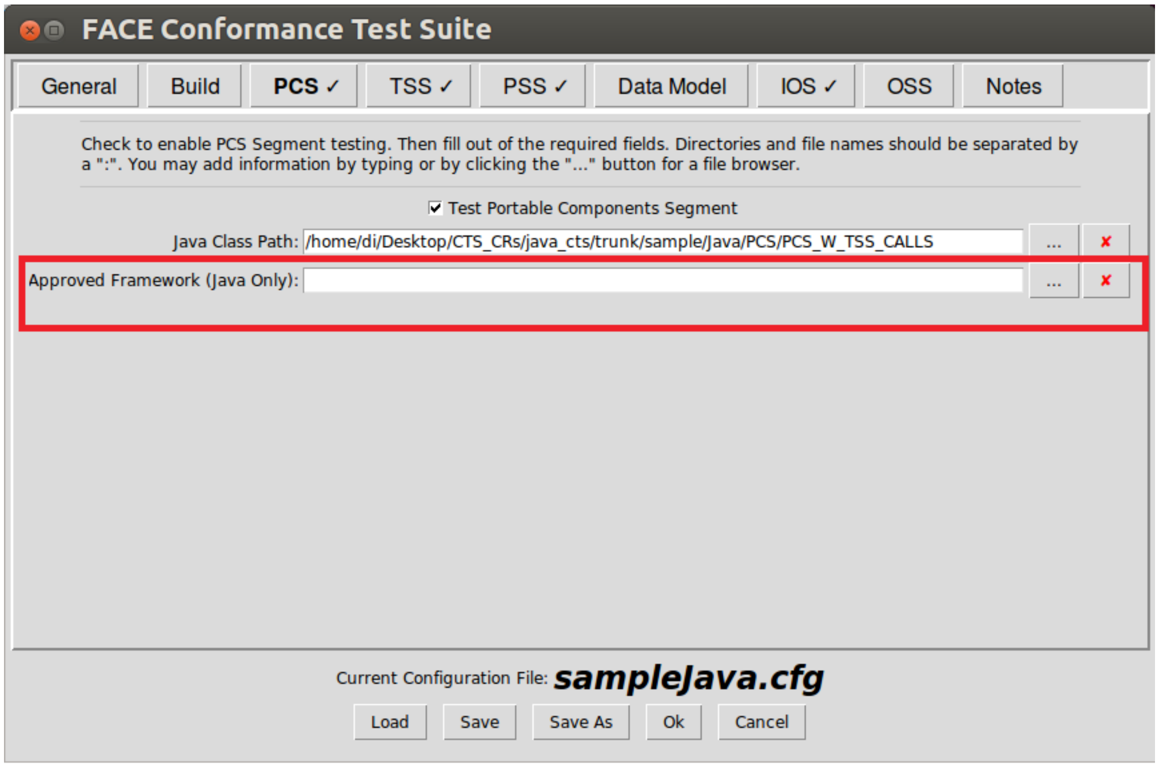
Use of Approved Frameworks

Approved Framework are classes that represent a third-party implementation of an OSS Interface not provided in the FACE Gold Libraries. Prior to version 2.1.0r8 and 2.1.1r4, these definitions are considered as a subset of the Supplier Class Definitions and provided by the supplier as a part of the “Java Class Path”. Starting with version 2.1.0r8 and 2.1.1r4, a GUI entry has been added to PCS, TSS, PSS, and IOS tabs of the configuration dialog box, see below figure. It has the same functionality as the “Java Class Path” entry that allows selection of directory(s) of .class and .jar files, or individual jar files.

Currently, the FACE Technical Standard makes provisions for one Approved Framework, OSGi, which is Java-based. OSGi is comprised of three specifications, but only the OSGi Core specification is allowed for FACE Conformance.

For verifying a UoC’s adherence to the OSGi Core API, provide the JAR file containing the interfaces defined in the OSGi Core specification found through the OSGi website (https://www.osgi.org/). This JAR file can be placed as an entry to CTS as an approved framework.

Note: Unless otherwise stated in an Approved Correction, the OSGi Core specification JAR file from OSGi is the only input allowed via the Test Suite’s approved framework mechanism. The third-party class definitions that provide a Component Framework must be proposed to and approved by the FACE Consortium before it can be used as an Approved Framework by CTS.



Check to enable PCS Segment testing. Then fill out of the required fields. Directories and file names should be separated by a ":". You may add information by typing or by clicking the "..." button for a file browser.

☑ Test Portable Components Segment

Java Class Path: /home/di/Desktop/CTS_CRs/java_cts/trunk/sample/Java/PCS/PCS_W_TSS_CALLS

Approved Framework (Java Only):

Current Configuration File: sampleJava.cfg

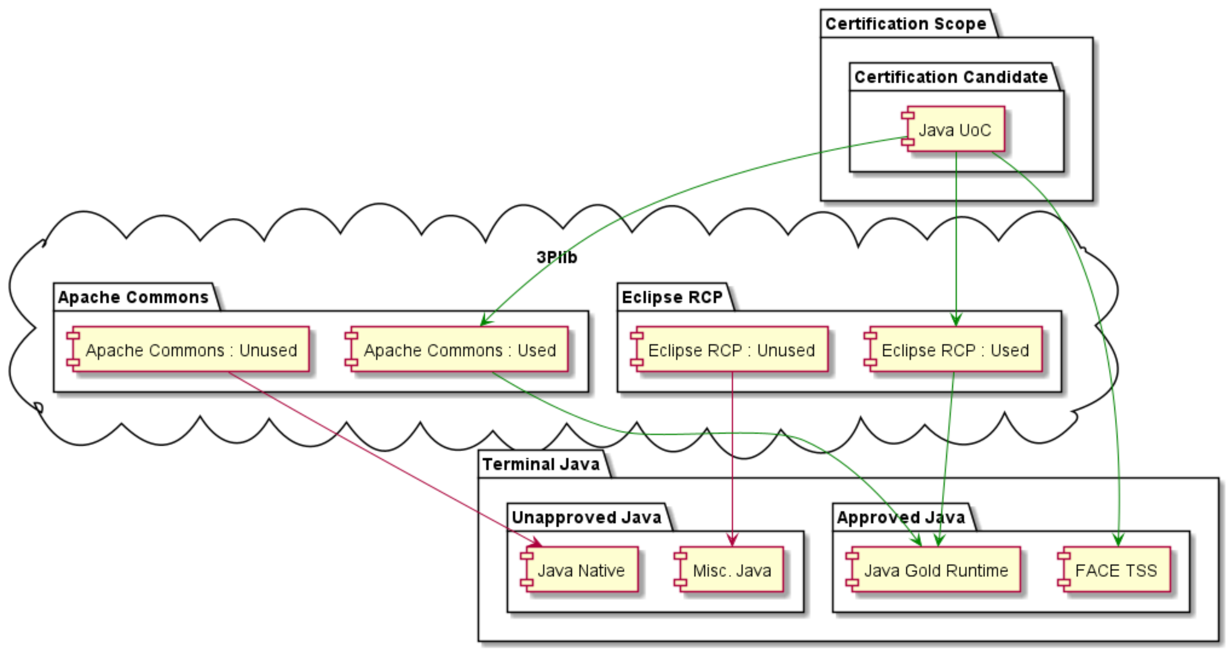
Approved Framework Entry – Java Only

Unused Code Paths in Third-Party Libraries

Typically, only a portion of the third-party libraries included for testing contain references needed to execute the UoC under test. To avoid unresolved reference errors from a third-party library's compile-time dependencies, it is acceptable to apply software tools to eliminate un-used code, e.g., ProGuard.

After applying the software tool, the resulting libraries are provided as part of the UoC. Additionally, a report generated from applying the software tools to remove the unused code should also be provided to the Verification Authority.

This approach shrinks the libraries provided to contain only dependencies needed by a UoC using the UoC as the set of entry-points. The ancillary calls and dependencies are eliminated leaving only calls and dependencies used by the UoC. The following figure is a high-level model showing a UoC where the green arrows represent the dependencies needed by the UoC for conformance testing and the red arrows represent dependencies in the third-party library not needed for testing the UoC.



Used and Unused References in Third-Party Libraries – Java Only

Considerations for Testing a Data Model

A Model Report Tool has been included with CTS that reads a data model (i.e. .face file) and produces a report about the model. The report is in the form of a PDF or HTML file. The purpose of the report is to provide insight into the model's consistency with a UoC's design for items the CTS cannot validate. This report may also provide insight into a UoC's use of the FACE Technical Standard, wherein a key intent is to model the interfaces in a detailed manner.

The Model Report Tool is intended to be invoked directly by a user and as such CTS does not invoke it. The tool can be found at **face_conformance_app/java_apps/FACEModelReport/** and the user guide is located at **face_conformance_app/java_apps/FACEModelReport/docs/**. Please see the user guide for details on how to use the tool.

Note: Windows 10 is the only supported operating system for the Model Report Tool. It may/may not run on CentOS 7.

Generating Gold Standard Libraries

You do not have to link to the FACE gold standard libraries (FGSL) outside of the test suite, but they may be useful to incorporate into your automated build system. You will need to compile your source code with the FGSL header files in order to run the conformance test(s) correctly. Otherwise, you will/may introduce compiler specific method calls that are internal to standard OS libraries headers. (See Theory of Operation above.)

The libraries generated in the goldStandardLibraries directory will get overwritten during each conformance test. This option will allow you to save the allowable gold standard libraries for the segments you have specified in the configuration in a user specified directory. The default is the test suite's sample directory. There is also a readme.txt file that is generated which will specify which include directories containing FGSL headers are allowed for your configuration's segments. This option is provided to aid the user in compiling and building against the FGSL outside of the test suite and is not needed to be pressed during conformance testing of segments.

What You Must Provide

- Your Segment(s) general and build configurations
- Segments which you wish to generate allowable Gold Standard Libraries to link against
- To generate FGSL For PCS, PSS, and TSS, the associated USM and SDM

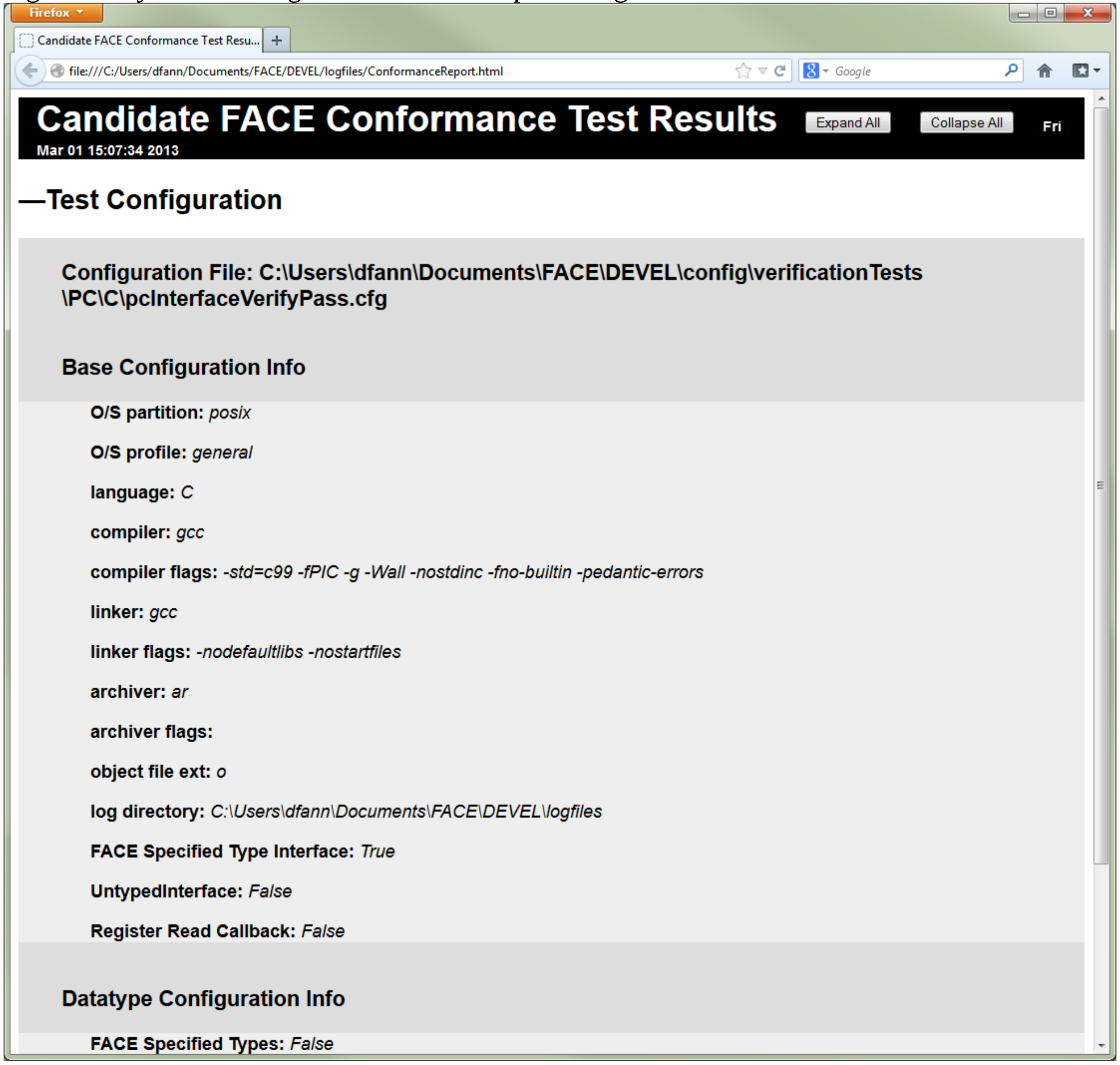
Procedure

1. Follow the procedure for testing a segment for each segment you wish FGSL generated. You may need to pick dummy segment object/header directories if they do not yet exist.

2. Click the "Generate Gold Standard Libraries" button on the main menu. Pick the directory where you wish the files to be generated.

Viewing Test Suite Results

Once the Test Segment button is pressed, the test suite will conduct the conformance test and launch a browser with the results in HTML format. The file will be called ConformanceReport.html in the log directory specified in the configuration. The resulting page also uses the ConformanceReport.css cascading style sheet and the ConformanceReport.js java script file as well as java script files in the scripts sub-directory and the style sheets in the styles sub-directory. All log files generated in the test will also be found in the log directory, although the same log files are found inside the HTML report file. Previous tests are completely erased with each test run and may also be deleted from the main menu "Delete Previous Test" button. You must make a copy if you wish to preserve a given test result. (Or alternatively, change the log directory in the configuration. An example configuration is shown below:


Firefox
Candidate FACE Conformance Test Resu...
file:///C:/Users/dfann/Documents/FACE/DEVEL/logfiles/ConformanceReport.html
Google
Expand All
Collapse All

Candidate FACE Conformance Test Results Fri Mar 01 15:07:34 2013

—Test Configuration

Configuration File: C:\Users\dfann\Documents\FACE\DEVEL\config\verificationTests\PC\C\pcInterfaceVerifyPass.cfg

Base Configuration Info

O/S partition: *posix*

O/S profile: *general*

language: *C*

compiler: *gcc*

compiler flags: *-std=c99 -fPIC -g -Wall -nostdinc -fno-builtin -pedantic-errors*

linker: *gcc*

linker flags: *-nodefaultlibs -nostartfiles*

archiver: *ar*

archiver flags:

object file ext: *o*

log directory: *C:\Users\dfann\Documents\FACE\DEVEL\logfiles*

FACE Specified Type Interface: *True*

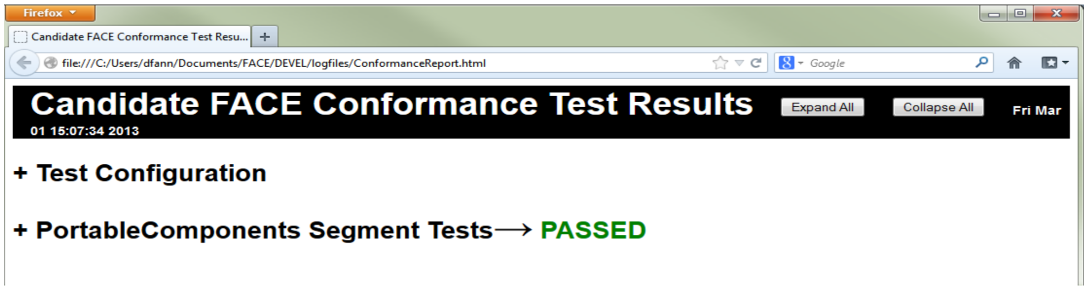
UntypedInterface: *False*

Register Read Callback: *False*

Datatype Configuration Info

FACE Specified Types: *False*

An example of a passed portable component is given in the figure below:

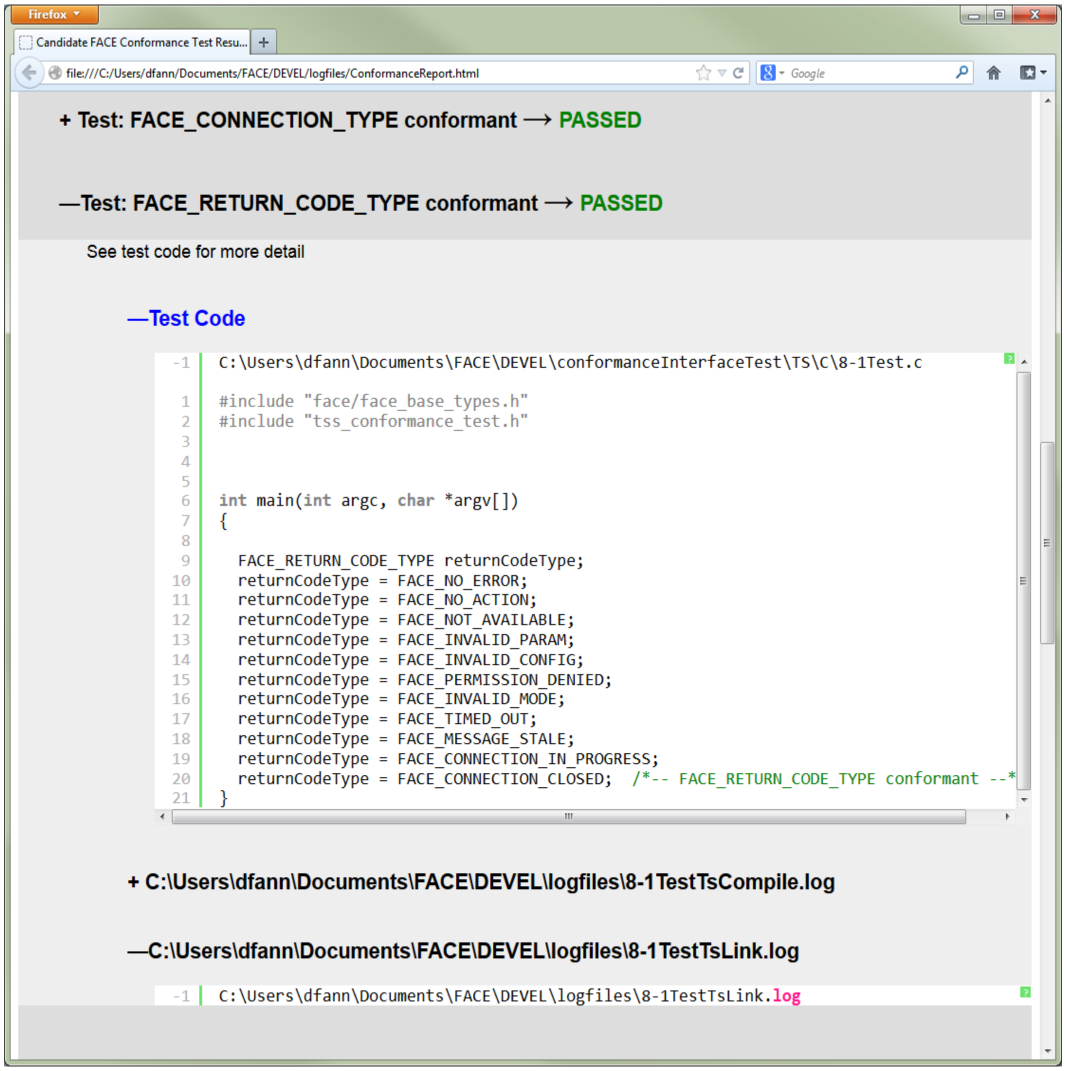

Firefox
Candidate FACE Conformance Test Resu...
file:///C:/Users/dfann/Documents/FACE/DEVEL/logfiles/ConformanceReport.html
Expand All
Collapse All

Candidate FACE Conformance Test Results Fri Mar 01 15:07:34 2013

+ Test Configuration

+ PortableComponents Segment Tests⟶ PASSED

To expand information on the configuration or test result, simply click on the line. If there is source code and/or log results associated with a test, they may be viewed by expanding the test results recursively. See examples of the expanded configuration and test results below.


Firefox
Candidate FACE Conformance Test Resu...
file:///C:/Users/dfann/Documents/FACE/DEVEL/logfiles/ConformanceReport.html

+ Test: FACE_CONNECTION_TYPE conformant ⟶ PASSED

—Test: FACE_RETURN_CODE_TYPE conformant ⟶ PASSED

See test code for more detail

—Test Code

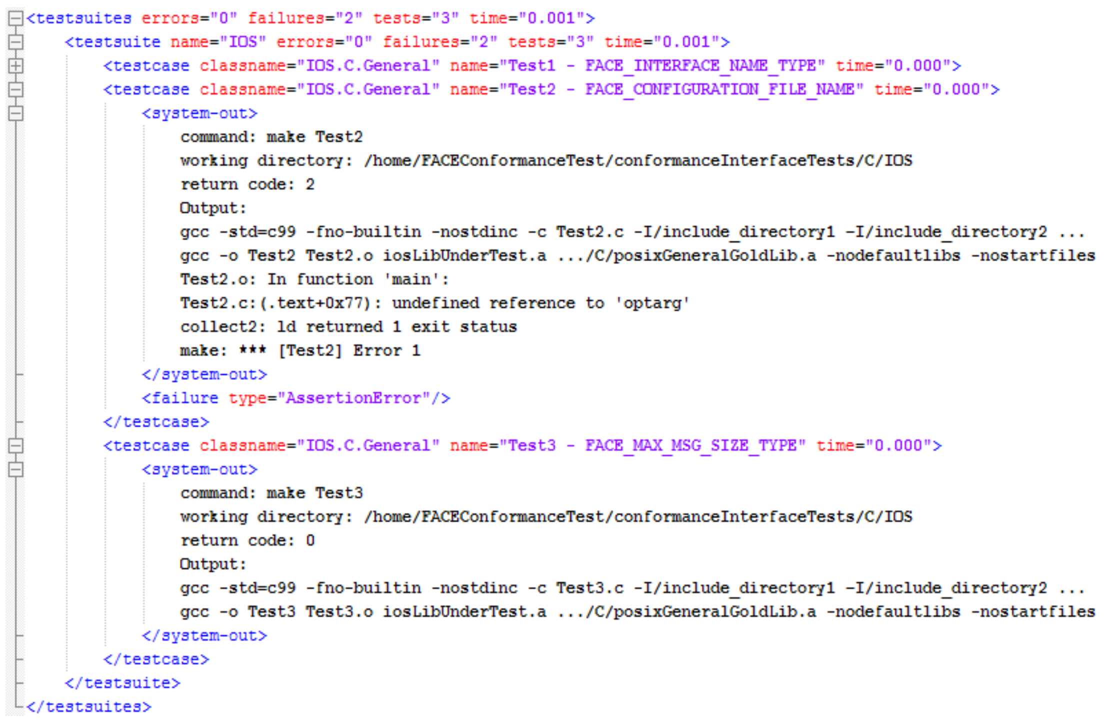
```
C:\Users\dfann\Documents\FACE\DEVEL\conformanceInterfaceTest\TS\C\8-1Test.c

#include "face/face_base_types.h"
#include "tss_conformance_test.h"



int main(int argc, char *argv[])
{

  FACE_RETURN_CODE_TYPE returnCodeType;
  returnCodeType = FACE_NO_ERROR;
  returnCodeType = FACE_NO_ACTION;
  returnCodeType = FACE_NOT_AVAILABLE;
  returnCodeType = FACE_INVALID_PARAM;
  returnCodeType = FACE_INVALID_CONFIG;
  returnCodeType = FACE_PERMISSION_DENIED;
  returnCodeType = FACE_INVALID_MODE;
  returnCodeType = FACE_TIMED_OUT;
  returnCodeType = FACE_MESSAGE_STALE;
  returnCodeType = FACE_CONNECTION_IN_PROGRESS;
  returnCodeType = FACE_CONNECTION_CLOSED;  /*-- FACE_RETURN_CODE_TYPE conformant --*
}
```

+ C:\Users\dfann\Documents\FACE\DEVEL\logfiles\8-1TestTsCompile.log

—C:\Users\dfann\Documents\FACE\DEVEL\logfiles\8-1TestTsLink.log

```
C:\Users\dfann\Documents\FACE\DEVEL\logfiles\8-1TestTsLink.log
```

For a failed conformance test, by examining the test source code and resulting log files, the source of the non-conformance should be able to be determined.
In addition to the HTML report, the test suite writes JUnit XML formatted files that are useful in automated build/test systems. An example XML file from a sample IOS test is included below. By using the command line version of the test suite described on the next page, conformance tests can easily be incorporated into an automated build process. The Junit XML files that are generated are more easily processed by integration software, such as Jenkins, than the more human readable HTML formatted file.



```
<testsuites errors="0" failures="2" tests="3" time="0.001">
    <testsuite name="IOS" errors="0" failures="2" tests="3" time="0.001">
        <testcase classname="IOS.C.General" name="Test1 - FACE_INTERFACE_NAME_TYPE" time="0.000">
        <testcase classname="IOS.C.General" name="Test2 - FACE_CONFIGURATION_FILE_NAME" time="0.000">
            <system-out>
                command: make Test2
                working directory: /home/FACEConformanceTest/conformanceInterfaceTests/C/IOS
                return code: 2
                Output:
                gcc -std=c99 -fno-builtin -nostdinc -c Test2.c -I/include_directory1 -I/include_directory2 ...
                gcc -o Test2 Test2.o iosLibUnderTest.a .../C/posixGeneralGoldLib.a -nodefaultlibs -nostartfiles
                Test2.o: In function 'main':
                Test2.c:(.text+0x77): undefined reference to 'optarg'
                collect2: ld returned 1 exit status
                make: *** [Test2] Error 1
            </system-out>
            <failure type="AssertionError"/>
        </testcase>
        <testcase classname="IOS.C.General" name="Test3 - FACE_MAX_MSG_SIZE_TYPE" time="0.000">
            <system-out>
                command: make Test3
                working directory: /home/FACEConformanceTest/conformanceInterfaceTests/C/IOS
                return code: 0
                Output:
                gcc -std=c99 -fno-builtin -nostdinc -c Test3.c -I/include_directory1 -I/include_directory2 ...
                gcc -o Test3 Test3.o iosLibUnderTest.a .../C/posixGeneralGoldLib.a -nodefaultlibs -nostartfiles
            </system-out>
        </testcase>
    </testsuite>
</testsuites>
```

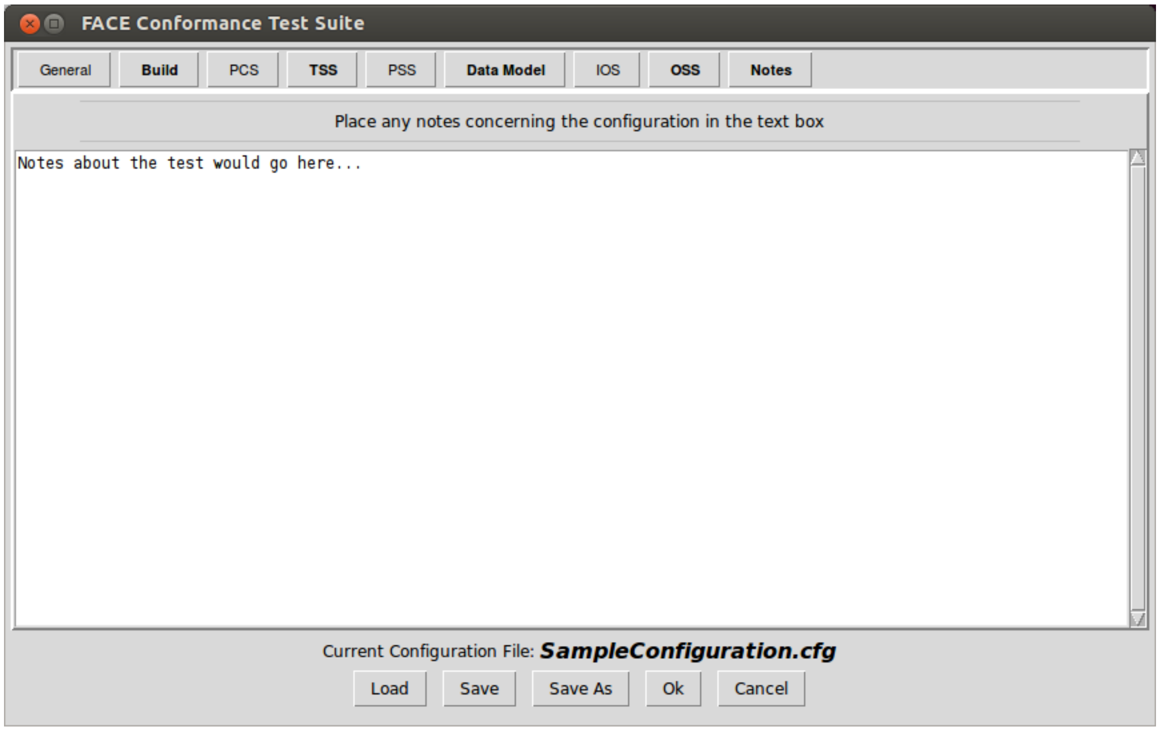
As shown, each test is recorded along with its name. Also, the JUnit report details the test segment, language, and profile as specified in the configuration file. Additionally, inside each test case node, the log file contents are included in a <system-out><\system-out> node in order to allow the inspection of the result of running each test. Also, in case of a test failure, a <failure><\failure> node is inserted for tests that fail the conformance test. Currently the JUnit test are generated for C, C++, and Ada tests.

Special Test Cases:

There are a few test cases that may result in an INSPECTION REQUIRED instead of the typical PASS/FAIL result. For PCS and PSS tests, a subsequent test occurs to determine if functions restricted to intra-UoP communication listed in section E-4 of the standard. If any of these function calls exist in a PCS/PSS, a code inspection is required. The second test case occurs for a safety extended profile for any PCS, PSS, IOS, or TSS. In this case, if a fork is called, then it must be followed by an exec, which requires code inspection. Neither of these tests are performed until the segment passes conformance without these restricted cases. For OSS, TSS and IOS tests performed using Ada, certain compile-time checks on the interfaces to be provided by these segments will only produce a compiler warning (not an error). To alleviate the Test Suite from assuming a specific vendor's compiler warnings, these tests are flagged to inspect the compiler output for warnings caused by the test's source code file.

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.


FACE Conformance Test Suite
General
Build
PCS
TSS
PSS
Data Model
IOS
OSS
Notes
Place any notes concerning the configuration in the text box

```
Notes about the test would go here...
```

Current Configuration File: ***SampleConfiguration.cfg***

Load
Save
Save As
Ok
Cancel

Test Suite Command Line Options:

There are a number of options you can use when running the test suite start-up script (runConformanceTest.sh on Linux runConformanceTest.bat on Windows)

Without any options or configuration files, the Test Suite GUI is launched with an initial configuration. The test suite comes with a default initial configuration file, but you can set your own by overwriting the initialConfigurationLINUX.cfg or initialConfigurationWINDOWS.cfg files in the face_conformance_app directory.

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 test.

The usage statement from the start-up script is shown below (Linux version, but options are same for Windows)

Usage: runConformanceTest.sh [options] [config_file1] [config_file2] ...

If config file(s) is(are) supplied, conformance test will run directly (no
GUI). If no config_file is supplied, the test suite GUI will launch for
interactive control of conformance test.

Options:

-h, --help	Show this help message and exit
-d new_project_directory, --projectDir=new_project_directory	Run test suite using a saved configuration file but changing the project directory to new_project_directory. Changing project directory only affects segment related files, not build files. A configuration file must be specified, and the original project directory must be specified in the configuration file. Multiple configuration files may be specified.
-m --model	Run data model test only.
-o new_log_directory, --logDir=new_log_directory	Run test suite using a saved configuration file but changing the log directory to new_log_directory.
-g, --gui	Run test suite with GUI even if a configuration file is supplied.
-v, --version	Echo test suite version and FACE Standard version against which conformance is checked.
-l generated_library_directory, --gsl=generated_library_directory	Generate Gold Standard Libraries needed by segments specified in the configuration file and store in the generated_library_directory. A configuration file must be specified, but subsequent configuration files will be ignored

Example Segments

The test suite has a sub-directory named “sample” that contains very simple examples of each segment in all four supported languages. Please refer to the readme in this directory for more information.

Known Issues

The Makefiles that are used to build both the gold standard libraries and conformance tests are flexible enough to handle most compilers without any alteration other than the configuration options that can be specified in the configuration menu. However, in extreme circumstances, it may be necessary to alter them for a unique build environment. The makefiles used for conformance tests are found in the conformanceInterfaceTests/LANGUAGE/SEGMENT directories. The makefiles used to build the gold libraries are in the goldStandardLibraries/LANGUAGE directories.

If system headers are used instead of gold standard headers in compilation of segment software under tests, 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 shown 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 1.0 of the conformance test suite and cannot be ported into 2.x conformance tests. The configuration file structure has changed slightly from previous versions of the 2.x conformance test but may be read by the current test suite. It will prompt you to save any old configuration files in the new format.

Acknowledgments

The test suite utilizes the following freely distributable software packages:

pyparsing 2.0.1
http://pyparsing.wikispaces.com/
Author: Paul McGuire
License: MIT License

stringtemplate 3.1
http://www.stringtemplate.org/
Author: Benjamin Niemann
License: BSD

Protocol Buffers - Google's data interchange format
http://code.google.com/p/protobuf/
Copyright 2008 Google Inc. All rights reserved.
License: New BSD

ANTLR
http://www.antlr2.org/
Copyright (c) 2003-2006, Terence Parr
License: BSD

References

The Java Virtual Machine Specification, Java SE8 Edition, Chapter 4
https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html