Architecture-based Unit Testing of the Flight software Product Line

Dharmalingam Ganesan[1], Mikael Lindvall[1], David McComas[2], Maureen Bartholomew[2], Steve Slegel[2], Barbara Medina[2]

[1] Fraunhofer Center for Experimental Software Engineering,
20740 College Park, Maryland, USA,
{dganesan, mlindvall}@fc-md.umd.edu

[2] NASA Goddard Space Flight Center (GSFC),
20771 Greenbelt, Maryland, USA,
{david.c.mccomas, maureen.o.bartholomew, steve.slegel, barbara.b.medina}@nasa.gov

Abstract. This paper presents an analysis of the unit testing approach developed and used by the Core Flight Software (CFS) product line team at the NASA GSFC. The goal of the analysis is to understand, review, and recommend strategies for improving the existing unit testing infrastructure as well as to capture lessons learned and best practices that can be used by other product line teams for their unit testing. The CFS unit testing framework is designed and implemented as a set of variation points, and thus testing support is built into the product line architecture. The analysis found that the CFS unit testing approach has many practical and good solutions that are worth considering when deciding how to design the testing architecture for a product line, which are documented in this paper along with some suggested improvements.

Keywords: unit testing, implemented architecture, mock, function hook, coverage, flight software.

1 Introduction

It is a well-known fact that the cost of finding and fixing a bug at the time of unit testing is cheaper than finding and fixing bugs that are found during integration testing, system testing or in the field. In addition, unit tests help developers while performing software changes because they indicate when changes break existing functionality. However, unit testing is not easy in practice for reasons including a) modules often depend on other modules, making them hard to separate and unit test in an independent fashion, and b) modules can also depend on unique features and functions provided by the operating systems, and they may require the hardware in-the-loop for the software to function properly, making it difficult to set up a controlled unit test environment. In the context of software product lines, one of the important concerns is the capability to unit test core modules without running and being dependent on the behavior of any other core modules, which might not be developed or correct at all times and for all possible scenarios. This capability is important

because the whole point of unit testing is to test an individual unit and to produce early and quick feedback regarding the test results.

In the context of product lines the situation is even more complex. For example, the core team must demonstrate the quality of their unit tests to the application team in order to build confidence regarding the quality of the core modules. Furthermore, when the application teams configure the variation points (e.g., features and modules to enable) of core modules or when they modify the source code of core modules, they need unit tests to help them quickly validate the correctness of the software. Because flight software is mission critical and needs to be of very high quality, the flight software branch at NASA GSFC has developed a practical approach for unit testing of its flight software product line (CFS). The Lunar Renaissance Orbit (LRO) mission which is currently orbiting the moon is one successful example usage of the cFE (core Flight Executive), which is the core of the CFS.

This paper discloses the architecture of the unit tests that are used in CFS, with the hope that other product line organizations may benefit from these ideas and concepts. The unit testing strategies described in this paper are sufficiently general and therefore also applicable to other product lines. The central ideas of the unit test architecture provided here include the ability to manipulate return codes of functions that are defined in dependent modules, used by the function under test. In fact, the CFS unit testing framework is designed and implemented as yet another set of variation points, and therefore testing is built into the product line architecture. Thus, the architecture supports plug-and-play of modules where modules can be bound to stub modules for testing, and each instance of a product line can be assembly of mock modules. This supports incremental unit and integration testing because when it has been determined that a certain application works as expected using the stubbed modules, the "real" modules can incrementally be added, one by one, and the same unit tests can be executed again with growing confidence in the final product variant.

The results of the analysis of the CFS unit testing strategy and collection of unit tests show that they share a common look-and-feel in terms of the way they set-up the tests, manipulate return codes of functions defined in other modules they use as well as how they set-up makefiles to run the unit tests. Furthermore, the dependent modules need not compile or run, thus this strategy provides early and quick feedback on unit test results of the module under test.

The analysis of test coverage shows that all publicly visible APIs have dedicated test programs, and many of the internal functions are indirectly tested through the test programs developed for public APIs. Thus, all functions of each core module can be unit tested automatically. For each configuration parameter, there is a dedicated set of unit tests that test the behavior of the relevant functions with respect to the boundaries of the values of individual variation points. The analysis also identified a few design problems from the unit testing point of view. One of the problems is that some functions return the same return code from different paths, making it difficult to determine whether or not the given test input data traversed the intended path of the function under test. This example demonstrates the importance of design for unit testing. Another problem is that some of the unit tests are lengthy due to the fact that they try to test more than one scenario inside one test function, making it difficult to trace back from test failures to the exact scenarios that failed. These problems are already added to the CFS issue tracking system and are being addressed by the CFS

team. An important premise of this statement is that the unit tests are considered an integral part of the product, and configuration is managed just like the source code.

Contributions of the paper. While the software product line community has a growing collection of articles related to modeling and managing variability, there are only a few practically inspired and validated technical papers focusing on unit testing in the context of software product lines. To this end, we hope this paper makes the following contributions:

1. A practical method for unit testing in the context of product lines. The method is derived from the way the CFS team implemented unit testing and examples for the CFS are used to explain the method.
2. A simple, yet effective approach for extracting and analyzing the architecture of the unit tests including a list of criteria was used for reviewing the unit tests and which can be used by other analysis teams.
3. An improved understanding of the relationship between software architectural design and unit tests. That is, an understanding of what makes unit testing easier or harder to develop and maintain. In addition, the paper demonstrates, using concrete examples, the importance of following architectural rules to facilitate unit testing.

It should be noted that while this paper focuses on unit testing, other important types of testing such as integration and system testing are also needed, and are briefly discussed at the end of the paper.

2 The CFS Product Line Architecture

This section introduces the CFS product line architecture as a context for understanding the architecture of unit tests. For CFS business goals and heritage, see [2]. The CFS has a layered structure. The top level layer has a catalog of reusable mission independent modules (a.k.a. applications), which may be used in one or more missions. Mission-specific modules (a.k.a. applications), i.e. they are only used in one mission, are also part of this top level layer. The second level layer (the Core Flight Executive (cFE) services layer) is the core of the CFS. The core layer offers several services, for example, the software bus module for inter-application communication, and the executive service module that manages the lifecycle of each application on the top level. Below the core layer, there is an OS abstraction layer (OSAL) which offers a common API for all operating systems supported by CFS (e.g. Vxworks, Rtems, and Unix), which was also released as open source [4]. There is also a board support package layer (BSP) which loads the configured OS and boots the CFS as well as a hardware abstraction layer, which offers a hardware-independent API for different types of hardware processors and ports, see Fig. 1. The cFE services and its lower layers are offered to various missions both inside and outside the NASA including a catalog of CFS applications that can be reused. Each cFE core service is configurable by choosing the values for appropriate constants declared in the interface or header files of each service.

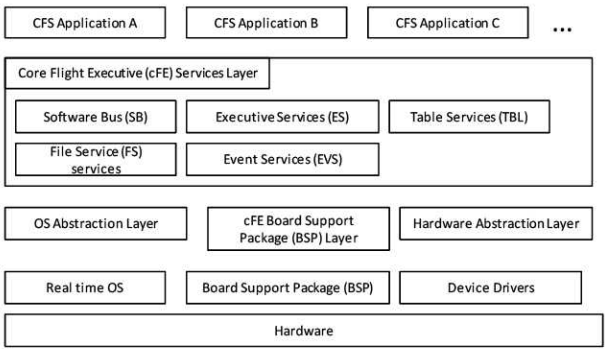


Fig. 1. The structure of the CFS Product Line.

All CFS modules are fully implemented in the programming language C. Each module has a set of C files with configuration parameters and public API functions declared in header files. There are dedicated makefiles for each module, which compiles all its files and produces an object file. All core modules are linked into one shared core library. Missions reuse this shared library and develop applications using the APIs offered by the core modules. Missions can add their own application modules to the top level application layer. However, in order to preserve the built-in flexibility and run-time reconfigurability, applications do not communicate directly with each other. Instead, applications communicate by subscribing to and publishing messages from the software bus and it is the responsibility of the software bus to deliver messages to all subscribed applications, see Fig. 2. The software bus is an abstraction built on top of OS queues and sockets making the applications unaware of the communication mechanism, which thus can be chosen at build time.

In [2], the CFS source code was analyzed with respect to its compliance to architectural rules. The detected violations of the architectural rules have now been removed and a new version of the CFS has been released. The previous analysis concluded that the CFS implementation is indeed consistent with the specified architecture. That is, layering is in place, and all CFS applications communicate only using the software bus. In this paper, we focus on the CFS' unit testing strategy and will explain the architecture of unit tests based on the software architecture of the CFS. The overall high level question is how we can test CFS-like product lines, which have to be of very high quality. The first step towards such testing is unit testing.

3. Technical Set-up and Process for Reviewing Unit Tests

This section introduces the approach followed for the independent review of unit testing strategies and their accompanying unit tests etc. For this paper, the review was applied in an independent way: the CFS team provided the artifacts to the Fraunhofer team, which has not been involved in any way with the development or testing of CFS, for review, feedback and recommendations for improvements. The Fraunhofer team used their reverse engineering and software architecture competency to review the unit tests of the CFS. It was the task of the Fraunhofer team to independently

understand how the unit testing is performed, and to identify issues with the current practice. These issues were then presented to the CFS team in technical meetings with the CFS engineers, project leaders, test leaders, etc. This process has been going on for 2 years and is reiterated whenever new releases or test suites have been developed.

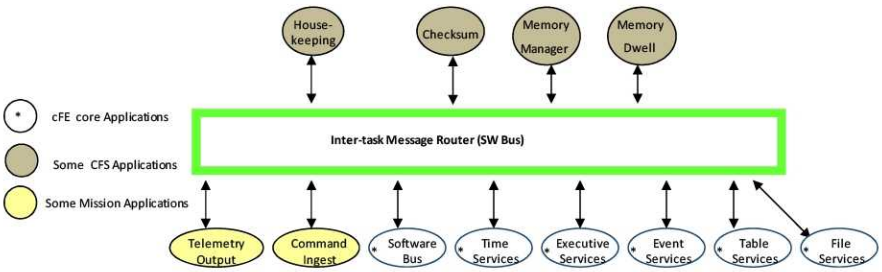


Fig. 2. The context diagram of the software bus in the CFS. Each module (a bubble) runs in a separate task, and communicates with other modules by publishing and subscribing to messages using the software bus.

The main goal of the review process is to get a good overview of the current state of the test suites. In order to do so, the review process attempts to formulate answers to the questions listed in Table 1. The approach to answering the questions is based on first extracting the architecture of unit tests from the test code. After that, the extracted unit test architecture was analyzed to understand the strengths and weakness. The practically relevant questions listed here are answered using a semi-automatic approach based on a reverse engineering and visualization tool suite used by the Fraunhofer team in projects with customers.

Table 1. Questions for reviewing the existing unit test code

Question	Purpose
1. Can core modules be tested independently of the core modules it uses?	To a) understand whether modules have unit tests, b) if there are architectural design issues that make unit testing hard.
2. How are variation points of each module being handled during unit testing?	To understand how to unit test the behavior with respect to each variation point (e.g. maximum number of messages in the software bus).
3. How easy is it to create mock or stub implementations of dependent modules?	To understand how complex it is to set-up so-called mock or stub implementations of dependent modules. Ideally, mock implementations are simple and their return values are easy to manipulate to traverse all paths.
4. Can modules be	To understand whether modules can be unit tested on

	unit tested without access to special hardware and/or OS?	standard desktop applications without requiring developers to access special hardware and real-time operating system.
5.	How easy it is to set-up the unit tests of a module?	To understand whether it is easy to set-up unit tests. Ideally, with a couple of instructions it should be possible to unit test a function.
6.	Are there dedicated test programs for each public function of a module?	To understand, from the coverage point of view: are there unit test programs developed to test each individual publicly visible API. From a reuse point of view, the trust increases if there are dedicated test programs for each API.
7.	How lengthy and complex is each test program?	To understand the complexity of unit tests. Ideally, unit tests focuses just on one scenario and do not mix multiple scenarios into one test program. Measuring the length and the number of conditional statements of each test program shed light on how well unit tests are structured internally.
8.	How are the test results collected and reported for further analysis?	To understand whether developers or testers can easily track back from test failures to the exact scenario. Are the code coverage results collected and stored either for further investigation or to derive new test cases.
9.	Is there a common look-and-feel in terms of the way the modules are unit tested?	To understand whether there is a well-defined architecture for unit tests, including constraints or rules for setting-up mocks, makefiles, set-up of tests and reporting of test execution results. Common look-and-feel a) helps programmers or testers to easily develop new unit tests, b) facilitates understanding of unit tests developed by different developers, and c) improves maintainability of unit test programs.

The Data Extraction Step: This first step involves parsing the existing source code and test suites to extract relations between entities (e.g. call relations, include relations) at the code level. The extracted relations are stored in a relational format in two databases: one database stores source code relations, and the other database stores relations of the test suite. This extraction is completely automated using parsers developed at Fraunhofer. The makefiles are also needed for the analysis because a) they contain information related to compiler switches, preprocessor symbols, and header files b) they contain information related to which object file is linked with the other object files. This linking knowledge is vital to extract correct dependency diagrams among modules, among test suites, and from test suites to source code modules. The ifnames tool is used to extract all conditional preprocessor symbols (excluding header file guards), which are basically variation points supported by the system. These variation points are later used to analyze how test suites handle them.

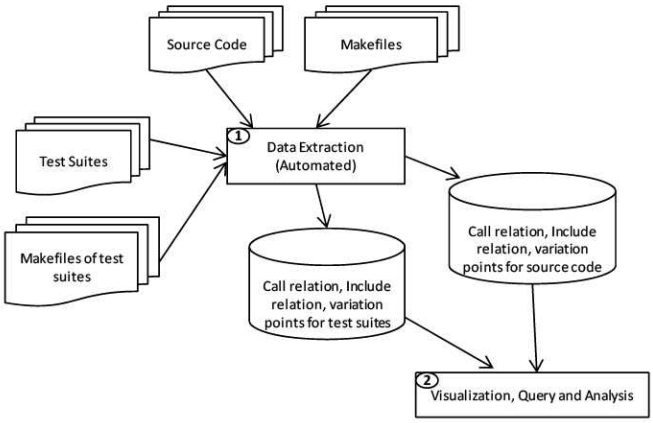


Fig. 3. Two major steps in the analysis of unit test architecture.

The Analysis, Query, and Visualization Step: In this step, the extracted data is analyzed using SQL-like queries written based on the RPA toolkit [1]. The RPA language supports several relation and set theoretic operators to query the extracted data. For example, it is possible to extract all functions defined in the source code which are not referenced by any of the test suites. Several RPA queries were developed for answering questions such as: 1) Is the given function tested at all by a test program or is the given function tested indirectly by a test program? 2) Is there is a stub or mock implementation of this function? 3) How many test programs call this given function, and 4) Which test cases refer the given variation point. While querying is useful to extract information, visualization is very powerful in revealing patterns in the structure of unit tests. Module level dependency diagrams and dependencies of test suites to source code modules were extracted using RPA and visualized using the SAVE tool, whereas the call graph of test suites are visualized using the Prefuse toolkit. The module dependency diagrams were used to review the structure of the test suites in terms of how dependencies to other modules are mocked.

4. Unit Testing of Core Modules

This section discusses the extracted architecture of the CFS unit tests based on the tool-supported process introduced in the previous section, see Fig. 4. Although it is almost a complete graph, only the offered public interfaces are used and no internal details of modules are shared with other modules. Also, there are clear reasons for each dependency. For example, the Executive Services (ES) module is responsible for initializing all modules, and all modules use the ES to register, create new tasks, or exit their execution. Similarly, all modules use the Software Bus to send and receive messages. The Event Service (ES) module helps modules log important events, and thus it is used by all modules. The File Service (FS) module helps modules write and read file data. The Timing Service module provides timing services to all modules. The Table Service (TS) module helps application layer modules register data tables to

share data with other modules. The dependencies were extensively reviewed by the CFS team members and all were deemed valid and necessary. The CFS' Interface Control Document (ICD), which is provided to application teams, specifies the interfaces of each core module. This English specification explains the behavior of each publicly visible function in terms of constraints on the input and the output.

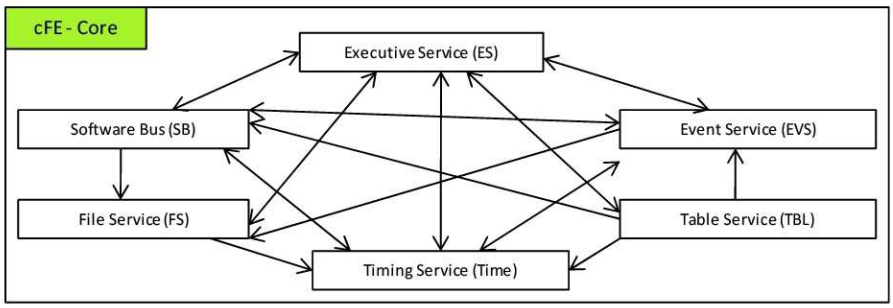


Fig. 4. Dependencies between core modules, extracted from the source code. Arrows represent code relations such as include, call, access of data structures, etc.

Variability in the Core Layer: As described in [2], there are only a few conditional preprocessor statements (e.g. #ifdef, #ifndef statements) in the core layer. For example, the timing service controls some variation points using #ifdefs, such as the Mission Elapsed Time (MET) and Greenwich Mean Time (GMT) formats. There are variation points in all modules, but instead of using #ifdefs, they are declared in the header files of the modules and can thus be configured individually for each mission. These variation points are basically constants. For example, the software bus has a variation point: maximum number of messages in the bus. Even though there are no #ifdefs, the cFE core can be executed on variety OS and hardware architectures because all modules of the core are programmed to the abstract interfaces of OS, hardware, and board support package abstraction layers. Given this brief overview of the architecture of the core layer and how variability is managed at the code level, the remaining section focuses on how each module is unit tested independent of other modules it uses. Fig. 5 shows an example of the high level unit test structure. This example view for the Executive Services (ES) module shows that the stub concepts are used in unit testing. For each core module, such a view was extracted from the test source code. It shows that the ES module depends on stub implementations of interfaces of the EVS, the SB, the Time Services, the Table Services, the File Services module, and also the stubs of OS and board support package APIs. This view is consistent with the source code dependencies of the ES module, shown in the previous Fig. 4, in that instead of using the real implementations of dependent modules, corresponding stubs are used. Note that stubs implement exactly the same interfaces that are implemented by real modules, and stubs are orthogonal – that is they are independent and don't need each other. At link time, the makefile of the module under test links its object files with the object files of stubs it uses. All stubs run in the same thread with the test suite. This analysis has shown that all core modules have the same high level structure as in Fig. 5, and that all their makefiles are customized in the same way in order to link to stub implementations of dependent

modules. Thus, they result in a good common look-and-feel in the high level structure of unit tests. Furthermore, developers can also replace stubs by real modules and can perform incremental module integration and validation.

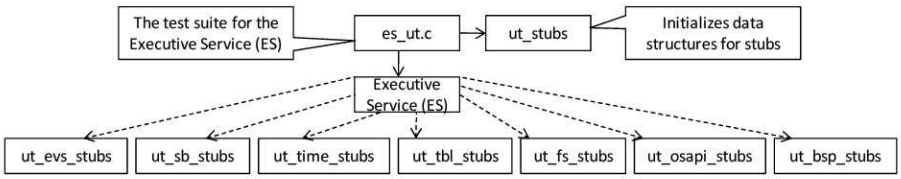


Fig. 5. The High level structure of Unit Tests (example view for the Executive Services (ES) module). Arrows denote dependencies (e.g. calls). Dotted arrows are dependencies established at link time. The ES module is linked to stubs that implement the interfaces of modules ES depends on. The main function is defined in es_ut.c (ES unit test) which runs all test programs implemented in es_ut.c.

Table 2. The number of stub functions used for testing each core module.

	Stub SB	Stub ES	Stub EVS	Stub Time	Stub TBL	Stub FS
SB	NA	11	3	1	0	1
ES	10	NA	4	3	1	4
EVS	8	10	NA	1	0	1
Time	9	8	2	NA	0	
TBL	9	15	3	1	NA	3
FS	0	2	0	1	0	NA

How the stubs are designed and implemented. The CFS implements stubs for each of the publicly visible APIs of its modules. The test suite for a specific module uses stub implementations of functions of other modules in order to fully run each function of the module under test (see Table 2) and in order to provide an environment that produces guaranteed results for each possible function call. In order to achieve 100% path coverage of each function under test, developers or testers also need a way to manipulate return values of the stubbed functions. Otherwise, unit tests will take a lot of time to run and it may also be difficult to pinpoint where a test actually failed. Keeping these requirements in mind, the CFS team has defined the data structure in Fig. 6 for unit testing purpose.

```
typedef struct
{
     uint32 count;
     uint32 value;
} UT_SetRtn_t;
```

```
void UT_SetRtnCode (UT_SetRtn_t *varPtr,int32 rtnVal,int32 cnt){
    varPtr->value = rtnVal;
    varPtr->count = cnt;
}
```

Fig. 6. The Key data structure used for controlling return values of functions. Each stub implementation of core module functions has its own instance of this structure. Testers manipulate the instance of this data structure. Stubs are programmed to return values of interest based on the state of the *count* variable. The logic of each stub is based on the state of the *count*. For example, a stub function can be implemented to return 0, if count is positive, and otherwise -1. Fig. **8** shows an example stub function. Right: Setting up the return values for each instance of the UT_SetRnt_t (in ut_stubs.c). The stub implementation for each function returns values based on the state of the count initialized using this function.

The ut_stubs module, shown earlier, creates several instances of the above data structure – one for each stub implementation of the core module functions. It is the responsibility of the tester to write stubs and manipulate return values using the state of *count* variable shown in Fig. 6. Note that all stubs have exactly the same function signature as the real the implementation. This is an important requirement otherwise the source code of the function under test has to be changed in order to unit test it, which is, of course, not a good engineering practice.

Consider the interface specification of the create pipe function of the software bus module, see Fig. 7. The original implementation returns one of four possible return values. However, the original implementation also creates real queues using the OS abstraction layer. If we want to unit test a function defined in another module that uses this create pipe function, the developer or tester should be given an easy way to manipulate return values so that different paths can be traversed easily. Also, in this scenario, the mock implementation does not need to create queues for unit testing of other modules. Such a mock implementation of create pipe is shown in Fig. 8. As we can see, it does not do too much in contrast to the original implementation. Nevertheless, it is remarkably useful from the testing point of view because of the capability it offers to control return values using the SB_CreatePipeRtn instance of the UT_SetRtn_t data structure.

```
/*****************************************************************************
** Name:    CFE_SB_CreatePipe
**
** Purpose: API to create a pipe for receiving messages
** Inputs:
**  PipeIdPtr - Ptr to users empty PipeId variable, to be filled by  this function.
**  Depth     - The depth of the pipe (max number of messages the pipe can hold at any time).
**  PipeName  - The name of the pipe displayed in event messages
**
** Outputs:
**   PipeId    - The handle of the pipe to be used when receiving messages.
**
** Return Values:
**    Status - CFE_SUCCESS, CFE_SB_BAD_ARGUMENT, CFE_SB_MAX_PIPES_MET, CFE_SB_PIPE_CR_ERR
**
******************************************************************************/
int32  CFE_SB_CreatePipe(CFE_SB_PipeId_t *PipeIdPtr, uint16  Depth, char *PipeName)
```

Fig. 7. Interface specification of the create pipe function of the software bus module.

```
extern UT_SetRtn_t  SB_CreatePipeRtn;

int32 CFE_SB_CreatePipe (CFE_SB_PipeId_t *PipeIdPtr, uint16 Depth, char *PipeName){
    if (SB_CreatePipeRtn.count > 0)
    {
        SB_CreatePipeRtn.count--;

        if(SB_CreatePipeRtn.count == 0)
        {
            return SB_CreatePipeRtn.value;
        }
    }
    return CFE_SUCCESS;
}
```

Fig. 8. The mock implementation of the create pipe function (in ut_sb_stubs.c file). This example shows how the UT_SetRtn_t data structure is manipulated to return different values.

Suppose we want to force the create pipe to return CFE_SUCCESS, all we need to do is just call the UT_SetRtnCode function as shown in Fig. 9 in our test function. This enables the test program to systematically control return values of other functions in order to traverse different paths of the program under test.

```
// forces CreatePipe to return CFE_SUCCESS
UT_SetRtnCode(&SB_CreatePipeRtn, -1, 2);
```

Fig. 9 Example of forcing a function to return the value of interest. See Fig. 6 (right) for the definition of this UT_SetRtnCode function.

The review has shown that all mocked functions and test programs follow this technique to manipulate return values. In addition, all mock implementations are very small, as little as 10 lines or so. Thus, it indicates that this technique works in practice and requires neither significant learning time nor major shift in the way of working. Table 3 shows that there are dedicated test programs for each public function of each core module. The two ES functions and one TBL function have no unit tests because they are single line get functions. The right hand side of the below table shows that not all internal functions are directly tested. However, further analysis has shown that they are transitively tested using the test programs of the public interfaces. This shows that the stub-based unit test architecture is possible to develop and works well in practice even though stubs and unit tests are manually developed at this point.

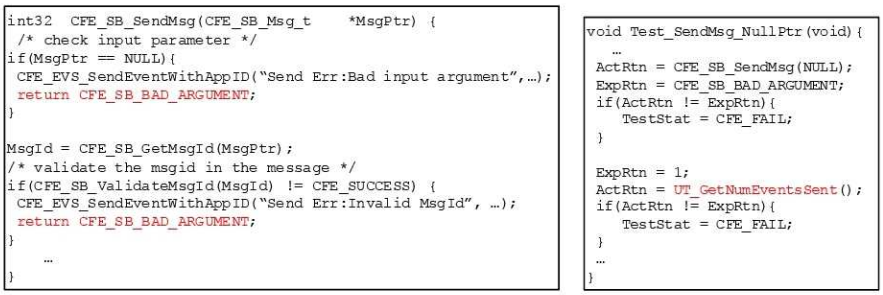
Table 3. Left: Interface coverage by unit tests. Right: The total number of functions unit tested directly. Some internals functions are also directly unit tested because they are defined as non-static C functions, otherwise internal functions are transitively tested using public APIs.

Core Module	# of Functions in Interface	# Directly invoked in Unit Tests
SB	30	30
ES	33	31
EVS	7	7
Time	24	24
TBL	14	13
FS	5	5

Core Module	# of Functions Defined	# Directly invoked in Unit Tests
SB	86	45
ES	117	68
EVS	33	12
Time	72	42
TBL	60	41
FS	11	11

Some design issues that make unit testing harder

Consider the code snippet defined in the software bus module (see the right of Fig. 10). It shows that the function returns the same value "bad argument" from two different conditional blocks. As a consequence, the unit testing code of this function becomes slightly more complex than necessary because it needs to determine exactly which one of the two code snippets returned that value. It does so by calling the stub implementation of the send event (similar to logging) function to make sure the number of times it was called is equal to 1 if MsgPtr is null, otherwise 2 if the msg id is invalid. This review identified a few functions that suffer from this design problem with respect to return values. These issues are being addressed by the CFS team. The recommended fix is to change such functions so that they all return a unique return value from each of its path, and thus make the unit testing code clearer.



```
int32  CFE_SB_SendMsg(CFE_SB_Msg_t    *MsgPtr) {
 /* check input parameter */
if(MsgPtr == NULL){
 CFE_EVS_SendEventWithAppID("Send Err:Bad input argument",…);
 return CFE_SB_BAD_ARGUMENT;
}

MsgId = CFE_SB_GetMsgId(MsgPtr);
/* validate the msgid in the message */
if(CFE_SB_ValidateMsgId(MsgId) != CFE_SUCCESS) {
 CFE_EVS_SendEventWithAppID("Send Err:Invalid MsgId", …);
 return CFE_SB_BAD_ARGUMENT;
}
    …
}
```

```
void Test_SendMsg_NullPtr(void){
   …
 ActRtn = CFE_SB_SendMsg(NULL);
 ExpRtn = CFE_SB_BAD_ARGUMENT;
 if(ActRtn != ExpRtn){
    TestStat = CFE_FAIL;
 }

 ExpRtn = 1;
 ActRtn = UT_GetNumEventsSent();
 if(ActRtn != ExpRtn){
    TestStat = CFE_FAIL;
 }
 …
}
```

Fig. 10. An example code snippet that makes unit testing difficult. It shows that the same return code is used for different issues. To unit test this function, the mock implementation of send event function counts the number of times the send event function is called, in order to make sure the correct path is tested for the test data. The right figure shows that the test case has to also test the side effect, that is, the number of logging events it was sent out by the function under test. UT_GetNumEventsSent gets the number of logging event using the data structure manipulated by the mock implementation of the SendEventwithAppId function, which simply counts the number of times it is being called.

Some design decisions that make unit testing easier.

Our review of unit tests has derived some insights on the influence of product line architectural design decisions on unit testing. Here, some product line specific examples from the CFS are disclosed.

The key to flexible unit testing is programming to abstract interfaces and moving out conceptually orthogonal variation points to the right module. For example, in the CFS case, the core layer is designed and implemented in such a way that it is completely agnostic to the OS, hardware, and board support packages. More concretely, consider the simple case of creating a queue, and sending and receiving messages using the queue. Naturally, different OSes offer different Queue APIs. If the system is programmed with a hard binding to the OS specific APIs then it is of course very difficult to unit test such a system, and a different sets of unit tests have to be developed for each OS type. In the CFS case, abstract interfaces with diversified implementations are developed, and thus conceptually orthogonal variation points are moved out of the module (see Fig. 11).

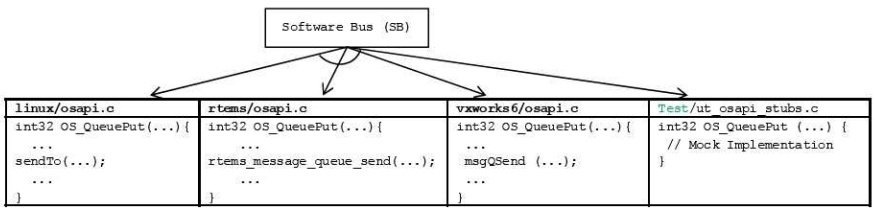


Fig. 11. A common abstract API with different implementations, including a mock implementation for unit testing. The SB module is programmed to the abstract interface, and the actual binding to a specific implementation is only at the link time.

Some internal details of modules should be made public. While hiding modules' secrets is one of the fundamental principles of software engineering [3], this principle has to be weakened in order to write good unit tests. For example, consider the load library function snippet (see Fig. 12), which loads the given shared library (LibName) and calls the function with the given name (EntryPoint). This is defined in the Executive Service module. The CFE_ES_MAX_LIBRARIES is a variation point defined in a public header file that must be set to a particular value. This function should return an error code if it is called more than the number of times set during configuration. Note that it uses the CFE_ES_Global data structure for keeping track of number of libraries that are already loaded. This data structure is hidden inside the ES module, meaning that no other module is allowed to access this data structure or know about it or its details. However, in order to test that this function will return an error code if it is called more than the configured number of times, the unit test must have access to CFE_ES_Global data structure; otherwise it is very difficult to simulate this error scenario. To this end, the CFS designers had made this global variable public to other internal files of the ES module, and thus the unit test can access and manipulate this variable. Architectural rules were defined to make sure such publicly visible secret variables are not referenced by other modules using the approach presented in [2]. This is an example of how the risk of violating some engineering principles can be mitigated by adding architecture/design rules.

```
int32 CFE_ES_LoadLibrary(char *EntryPoint,  char *LibName, …) {
   boolean LibSlotFound = FALSE;
   for ( i = 0; i < CFE_ES_MAX_LIBRARIES; i++ ) {
      if ( CFE_ES_Global.LibTable[i].RecordUsed == FALSE ) {
          LibSlotFound = TRUE;
          break;
      }
   }
   if(LibSlotFound == FALSE) return CFE_ES_ERR_LOAD_LIB;
}
```

Fig. 12: The Load library function loads the library (LibName) and calls a function (EntryPoint) of that library. CFE_ES_MAX_LIBRARIES is a variation point defined in a header file. CFE_ES_Global is a global variable, allowing the unit test to change the state to validate the scenario that if this function is called more than the configured number of times, an error will be returned code (see Fig. 13).

```
/* Test for loading more than max number of libraries */
 for (j= 0; j < CFE_ES_MAX_LIBRARIES; j++) {
        CFE_ES_Global.LibTable[j].RecordUsed = TRUE;
 }
 Return = CFE_ES_LoadLibrary("EntryPoint","LibName", …);
 UT_Report(Return == CFE_ES_ERR_LOAD_LIB, "CFE_ES_LoadLibrary",
                       "No free   library slots");
```

Fig. 13. Test code from the load library function that tests the behavior of the load library function when it is called more than the allowed number of times (CFE_ES_MAX_LIBRARIES) (see Fig. 12). It manipulates the ES module's internal data structure that keeps track of the number of loaded libraries.

Table 4 Answers to questions based on the analysis

Question	Answer and Comments
1. Can a core module be tested independently of core modules it uses?	Yes. Because of the novel design of simple stubs, it only takes 3 minutes or so to run all the unit tests of the core modules.
2. How are variation points of each module being handled during unit testing?	There is a unit test program for each variation point that checks the behavior for upper and lower bound constraints. Some internal details of a module are made public to support unit testing.
3. How easy is it to create mock or stub implementations of dependent modules?	Mock or stub implementations are easy to create. At link-time, a module can be linked to one or more stubs of dependent modules. This capability supports incremental integration too.
4. Can modules be unit tested without access to special hw and OS?	Yes. Testers can test on their desktop and do not need to go to the test lab for unit testing. The UTF framework provides simulators with the same API as the original code.
5. How easy it is to set-up unit tests for a module?	Just a couple of instructions are needed to set-up a test program.
6. Are there dedicated tests for each public function of a module?	Yes, all interfaces have one or more dedicated unit test programs.
7. How lengthy and complex is each test program?	Some are lengthy (~100 lines) because they test more than one scenario and could be split into smaller ones. Some are complex because the function under test returns the same return code from multiple paths requiring extra test code.
8. How are the tests results collected and reported for further analysis?	Currently, they use the gcov (GNU coverage) line coverage tool. All failures are reported in a text file that is manually reviewed by the tester.
9. Is there a common look-and-feel in the way the modules are unit tested?	Yes. All core modules consistently use the concept of stubs to do unit testing. Also, all test makefiles for test suites share the same structure.

5. Closing Remarks

In this paper, we described the analysis of the CFS product lines' unit testing strategy and accompanying unit test cases and testing environment. The CFS has been refined over more than 10 years and has gone through rigorous inspections and improvement initiatives. In addition, the CFS captures knowledge from implementing dependable flight software for more than 20 years of specifying, developing, testing and flying such software. Thus, we are grateful that we can analyze and use CFS as an example of good software engineering that we can all learn from, even though there are still some issues that can be removed and improved. For example, the CFS has tackled the

difficult practical unit testing problem that modules often depend on other modules, making them hard to separate and unit test in an independent fashion. In addition, modules can also depend on unique features and functions provided by the operating systems, and they may require the hardware in-the-loop for the software to function properly, making it difficult to set up a controlled unit test environment. The CFS team's approach to unit testing also handles the use of modules (real modules or stubs for testing) as a set of variation points. This introduces a level of flexibility that allows the user of CFS to also use the same set up for incremental integration testing because stubs for testing can be swapped in or out depending on the situation, thus limiting the risks that are associated with big bang integration testing. A future paper will disclose the unit testing framework that allows application developers to unit test their applications without running the core modules. Unit testing and the type of incremental integration testing described above are only two aspects of testing, and other forms of testing needs to be conducted in order to detect those types of defects that such testing cannot detect. Supported by the NASA IV&V center, Fraunhofer, in collaboration with the CFS team and using CFS as a testbed, are researching ways to develop new testing techniques that address these challenges.

Acknowledgments. Lisa Montgomery and her NASA IV&V team and Sally Godfrey NASA GSFC for supporting this work; Charles Wildermann, all members of the GSFC CFS team for comments and discussions; Rene Krikhaar for the RPA toolkit; The Prefuse visualization team, at Stanford University, for making it available to us; Lyly Yonkwa for fruitful discussions; three anonymous reviewers for comments.

References

1. Feijs, L., Krikhaar, R., and Van Ommering, R.: A Relational Approach to Support Software Architecture Analysis. Software Practice and Experience, 28(4):371-400, 1998.
2. Ganesan, D., Lindvall, Ackermann, C., M., McComas, D., and Bartholomew, M.: Verifying Architectural Design Rules of the Flight Software Product Line. SPLC, 2009.
3. Hoffman, D., Weiss, D.: Software Fundamentals – Collected Papers of David L. Parnas. Addison-Wesley Publications, 2001.
4. The OS Abstraction Layer of the CFS, http://opensource.gsfc.nasa.gov