Verifying Architectural Design Rules of the Flight Software Product Line

Dharmalingam Ganesan, Mikael Lindvall, Chris Ackermann
Fraunhofer Center for Experimental Software Engineering, Maryland
{dganesan, mlindvall, cackermann}@fc-md.umd.edu

David McComas, Maureen Bartholomew
NASA Goddard Space Flight Center, Greenbelt, Maryland
{david.c.mccomas, maureen.o.bartholomew}nasa.gov

Abstract

This paper presents experiences of verifying architectural design rules of the NASA Core Flight Software (CFS) product line implementation. The goal of the verification is to check whether the implementation is consistent with the CFS' architectural rules derived from the developer's guide. The results indicate that consistency checking helps a) identifying architecturally significant deviations that were eluded during code reviews, b) clarifying the design rules to the team, and c) assessing the overall implementation quality. Furthermore, it helps connecting business goals to architectural principles, and to the implementation. This paper is the first step in the definition of a method for analyzing and evaluating product line implementations from an architecture-centric perspective.

Keywords: business goals, architectural rules, implemented architecture, flight software

1 Introduction

It is a well-known fact that the software architecture is critical to the success of a software product line. This is reflected in the fact that organizations often spend significant effort on the design of the software product line architecture and strive to chose the most beneficial architectural styles, patterns, and decomposition strategies (e.g., [2],[3][9],[11]). These architectural design decisions are made to efficiently establish the core for a family of products, by taking advantage of their commonalities and carefully managing variability. One important factor determining the success of a product line is whether the "designed" variability is indeed present and maintained in the implementation. Thus, the challenge is to verify that the resulting implementation is consistent with the intended architectural design principles (e.g., [10],[12]).

There are a number of reasons why the implementation might deviate from the architecture, including a) the architecture is an abstract entity not directly expressible using standard programming languages, b) the architecture is typically not documented well enough for developers to be able to fully comprehend and follow during the implementation, c) performance and other non-functional issues, not easily detectable during architectural design time, have to be resolved with code-level workarounds, d) complexity in managing source code level variation points. These characteristics make it difficult and tedious to ensure that architectural principles are met through traditional inspections and reviews.

The Flight Software Systems Branch at NASA Goddard Space Flight Center (GSFC) has spent considerable resources over the past few years developing the Core Flight Software (CFS) and positioning it as the future flight software platform for NASA missions. The CFS follows a product line approach with the goal to support systematic reuse. The business goals of the CFS are the main drivers for creating and maintaining the product line architecture. Consequently, many of the architectural decisions are directly influenced by the business goals of the CFS. Thus, the CFS team needs to ensure that the business goals and the implementation are aligned by verifying that the implementation indeed follows the architectural principles, and the team has been doing so as part of rigorous code reviews.

The CFS developer's and deployment guide specifies the architecture principles in terms of structural and behavioral properties of the CFS product line architecture. Thus, it is possible to derive various types of architectural rules from the architecture principles (e.g. from the architectural principle of layering the rule that a lower layer cannot use an upper layer can be derived). The scope of this paper is the subset of rules that are related to the module structure (i.e. the module architecture [11]) of CFS and its development environment (i.e. the code architecture [11]), and that can be verified statically (i.e., without executing the system). The derived rules are categorized into module dependency-restriction rules, decomposition-restriction rules, redundancy-restriction rules, and miscellaneous rules, including visibility-restriction rules, conditional preprocessor directives-usage rules, and interfaces usage rules. These rules directly and indirectly address various concerns, such as run-time adaptability, portability, testability, and performance of the product line.

The analysis and verification of these architectural rules are conducted against the most recent source code version of the CFS product line, which includes the modules of the core framework and a set of optional reusable application modules. The results of the verification of these rules show that most of them are indeed followed in the implementation. Naturally, some deviations were detected through this process, and the high-priority issues are currently being addressed.

Discussions with the CFS team revealed that the software is written and reviewed by experienced engineers who have been developing flight software for about 15-20 years. Nevertheless, the team believes that this tool-supported independent verification of architectural rules complements such reviews because it identified architectural deviations that escaped the manual review process. Thus, it helps to preserve and protect the carefully designed variability points, with the effect that it increases the confidence of the overall product line quality. Furthermore, the analysis helps in establishing an explicit mapping among business goals, architectural principles, and implementation decisions.

This paper is the first step in the definition of a method for analyzing and evaluating product line implementations from an architecture-centric perspective. Complementary analysis of rules that are outside the scope of this paper (e.g. rules related to the behavior of the system) will be added to the method in the future. Contributions of this paper are: 1) a demonstration of how a collection of architecturally-relevant rules can be verified and analyzed in order to make sure the implementation is aligned with business goals, 2) data and insights from a product line implementation contributing towards developing a benchmark for evaluating the implementation quality of product lines.

2 The CFS Product Line

In this section, the heritage and business goals of the CFS product line are briefly discussed. In order to illustrate the importance of verifying the architecture rules against the implementation, the relationship between business goals and architectural rules is presented.

2.1 The Heritage of the CFS

In the past, when developing a new mission, the Flight Software (FSW) lead for the mission would obtain existing FSW and artifacts from heritage missions that they knew well (see figure 1). As the figure shows, the FSW branch at NASA GSFC has several "heritage architectures" to choose from. Once a fitting heritage mission had been selected, changes were made to the software artifacts in order to implement the requirements of the new mission. In addition, changes in the flight hardware or changes in the operating system caused changes throughout the software system. This ad hoc reuse implied that, for example, integration of new modules required extensive manual coordination. The conclusion was that the model based on reuse of selected heritage architectures was not flexible enough for collaboration within GSFC, with other NASA centers, or with outside entities. In addition, this reuse model forced the on-orbit FSW maintenance team to understand, in detail, each heritage architecture and its implementation because of the differences between the missions. Thus, cost advantages from this type of reuse were not visible.

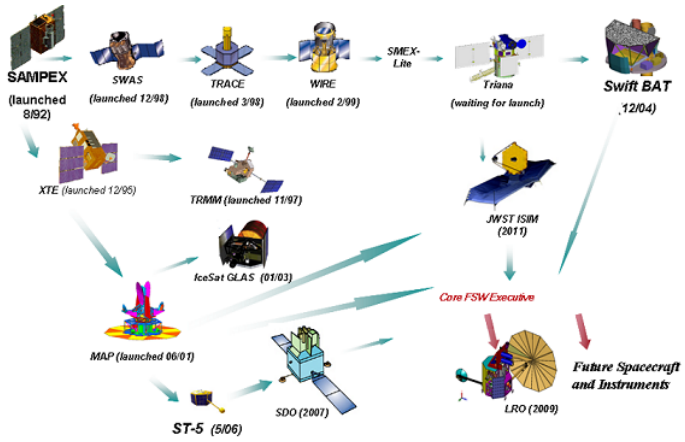

Figure 1 The CFS Heritage

Prior to the year 2000, FSW was developed in multiple branches. A single FSW branch was established in the year 2000 that created the foundation for a new FSW product line as a response to the software reuse problem. A heritage analysis, among the past missions, as illustrated in figure 1, showed that the requirements for Command and Data Handling (C&DH) Flight Software are indeed very similar from mission to mission. This heritage analysis was the starting point for the GSFC FSW branch's establishment of guidelines for the CFS product line, with the primary goal of not "re-inventing the wheel" in each mission project. In 2003, the development of the flight software product line started. The CFS is now used by several FSW projects.

2.2 Business Goals and Architecture of the CFS

The CFS product line is being developed to achieve the following business goals: a) reduce time to deploy high quality flight software, b) reduce project schedule and cost uncertainty, c) enable collaboration across organizations, d) simplify sustaining engineering, e) establish common standards and tools across the FSW projects and NASA wide, f) establish the use of a single platform for advanced concepts and prototyping, and g) directly facilitate formalized software reuse.

In order to achieve these business goals, the CFS team developed a software product line architecture based on solid software engineering principles, such as abstraction, information hiding, and modularity [8]. A layered architecture that hides the internal details of OS and hardware platform has been defined [6]. Core modules configurable for mission-specific needs were developed for reuse, forming the core (also called the "executive") layer of the CFS. A set of optional reusable modules (also called application modules) were also developed on top of the core layer. These application modules are optional, meaning that they need not be used in all missions. Mission-specific functionalities are introduced by plugging-in modules into the application layer. A detailed API specification explaining how to use the core or generic modules, (shown in the executive layer in figure 2) was also documented. Run-time module registration mechanisms have been created for integrating modules with little human effort. The CFS facilitates this mechanism by using a publish-and-subscribe architectural style with a software message bus (see figure 2) as the middleware.

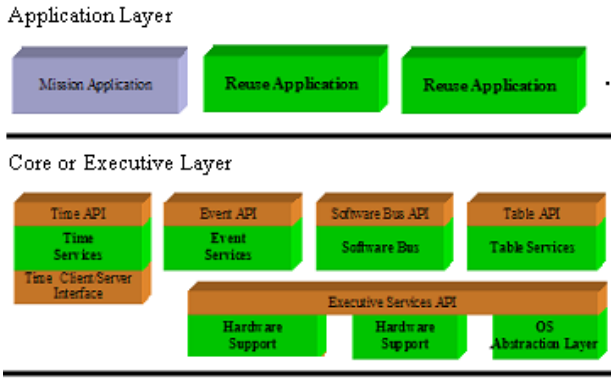

Figure 2 High-level structure of the CFS product line

In fact, there is a many-to-many mapping between the business goals of the CFS and the architectural decisions that were made (see figure 3). Thus, in order to meet the business goals, it is vital to verify that the implementation is consistent with the architecture rules, and resolve discrepancies, if any.

The interest in the CFS has been spreading fast within the aerospace community. For example, Lunar Reconnaissance Orbiter (LRO), Global Precipitation Measurement (GPM), Magnetospheric Multi-Scale (MMS) at NASA GSFC, Radiation Belt Storm Probes (RBSP) at Johns Hopkins University/Applied Physics Laboratory (JHU/APL), and Lunar Atmosphere and Dust Environment Explorer (LADEE) at NASA AMES, are all using the CFS, and many more are expected in the near future.

The modularity of the architecture also supports the funding model. The CFS relies on project and project-independent funding. For example, the GPM project is helping to fund the CFS application modules development. Future missions are expected to contribute to new capabilities development as well as to support CFS sustaining engineering.

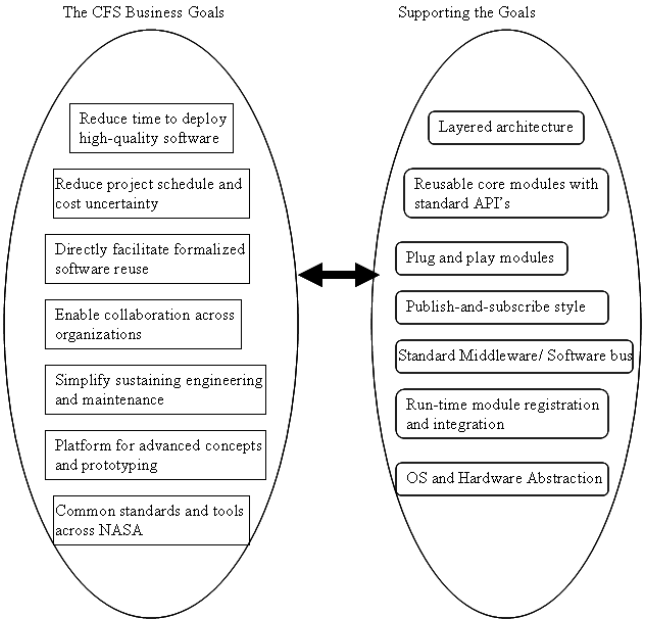

Figure 3 Business goals and architecture principles

3 Verification of Architectural Rules

3.1 General Process for Verification

The verification process involves two teams: the CFS team and the Fraunhofer team. The process started in early 2008 and has now been ongoing for more than one year. In order to perform this verification, the CFS team sends the requirement specification, the developer's and deployment guide, and the source code bundle to the Fraunhofer team. All core modules (also called services) and the application modules developed by the CFS team are part of the source code bundle. Figure 4 shows an example CFS context. The core services (in green color) and applications (in blue color) of figure 4 correspond to the core layer and the application layer of Figure 2, respectively. Mission-specific application modules (in yellow color) are not included in the verification process. The Fraunhofer team then performs an independent verification of architectural rules using their reverse engineering methods and tools. After the analysis, both teams get together for detailed discussions on the results of verification. These meetings often lead to follow-up analysis as new questions arise. After each meeting, the CFS team addresses the high-priority architectural issues, and the Fraunhofer team prepares answers for the additional questions raised by the CFS

team. This process is repeated periodically based on the progress of the CFS development, for example, after the implementation of new application modules. This verification task is non-intrusive and non-biased, because on the one hand it does not affect the CFS team's development process, and on the other hand the analysis is independently performed by an external organization (i.e., Fraunhofer).

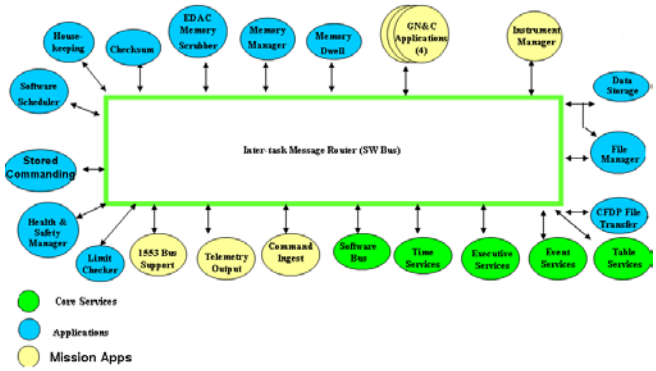

Figure 4 The CFS example context

3.2 Overview of the Approach

The steps followed for verifying the architectural rules are depicted in figure 5. Here, these four major steps are briefly explained.

Step 1 - Derive Architectural rules: The CFS requirement specification describes the functional requirements addressed by the core modules. The application guide describes, in detail, the APIs of the core modules. Furthermore, the way the core modules should be used by application modules are also described in this document. In addition, the application guide contains samples demonstrating how to develop and integrate new application modules with the core modules. The deployment guide explains how to deploy the CFS for individual missions. From these documents, it is possible to derive a number of architecturally-significant rules. This manual step has to be conducted only once since the architectural rules do not change frequently as compared to, for example, source code. A fragment of these rules and the related quality attributes they address are placed in Table 1. Note that the association between rules and quality attributes is applicable for the CFS and might differ for other contexts. The absence of a quality attribute for a rule does not necessarily imply that the attribute is irrelevant. Only the highly relevant quality attributes are associated to each rule in the table. The output of this step is a collection of architectural rules. In the future, the CFS team will offer the collected architectural rules to the CFS team as well as to other teams that use the CFS in their missions. It is expected that this will create better awareness and a clarification of the relationship between rules and quality attributes.

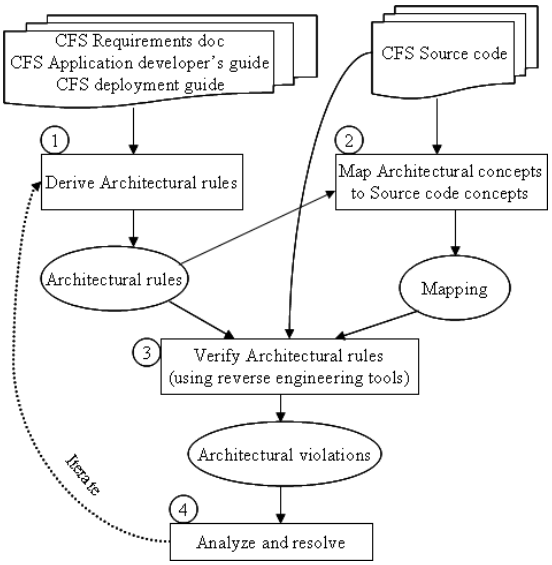

Figure 5 Four Steps in architectural rules verification

Rule Type	Rule Scope	Highly Related Quality Attribute
Dependency rules	1.Do generic modules depend on specific modules? 2. Do modules bypass the OS and hardware abstraction layer? 3. Do modules follow the publish-subscribe architectural style?	Run-time adaptability, Buildability, Portability, Testability, Performance
Decomposition rules	1.Do modules directory structure adhere to the template structure? 2.Do modules follow the decomposition guidelines?	Buildability, Comprehensionability
Redundancy-restriction rules	Do modules copy-and-paste from other modules?	Maintainability
Miscellaneous rules	1.Do modules expose their internal details? 2.Do modules implement necessary interfaces, for example the logging interface?	Changeability

Table 1 Sample Rules and related quality attributes

Step 2 – Map Architectural Concepts to Source Code Concepts: The derived architectural rules are abstract and not necessarily explicit in the source code. For instance, the concept of the OS abstraction layer is architectural, and the corresponding source code concepts need to be clarified for developers to be able to use them. The CFS documents provide such mapping,

for example, they describe which directory of the CFS implements the OS abstraction layer.

Architectural Concept	Source Code Concept
Layer	Directory
Module	Sub-directory
Interface	Function Signature

Table 2 Architecture and source code concepts

Similarly, architectural concepts, such as *application layer*, *core (*or *executive) layer*, *middleware* and *software bus* are also explained with a mapping to source code concepts, such as directories, files, and functions (see Table 2).

Step 3 – Verify Architectural Rules: In this step, the source code of the CFS is verified with respect to the architectural rules using the mapping defined in the previous step. Tool support is needed because a) the source code is too large for manual review, and b) whenever the source code changes, verification needs to be performed again. A few tools are employed for verification. First, the Fraunhofer SAVE tool [13] has the capability to easily import the source code and extract code relations as shown in Table 3. The tool offers a GUI to define the mapping and graphical editors for specifying the architectural rules. Using regular expressions, it is possible to define a map, for example, that all files under cfs-apps directory are part of the application layer. Other tools, such as the Relation Partition Algebra (RPA) tool [4] [10] are used to complement the SAVE tool, for example to verify rules related to interface usages and module visibility-restrictions.

Relation Type	From	To	Comments
Call	Function	Function	The *Call* relation is from a caller to a callee
Include	File	Header file	The *Include* relation is between a file and a header file
Refer variable	Function	Variable	The *Refer_Variable* relation is between a function and a global variable referenced by that function
Part_of	Function, Variable, File	File, Directory	The *Part_of* relation models the hierarchical decomposition. For example, functions are part of files, whereas files are part of directories.

Table 3 Sample relations extracted from code

The output of this verification step is a collection of inconsistencies between architectural rules and the source code.

Step 4 – Analyze and Resolve: The focus of this step is to analyze and resolve the architectural deviations. Depending on the criticality of the deviation, these deviations are resolved at the source code-level. In certain cases, the deviations are exceptions to architectural rules, and need not be resolved. This process is iterated either when new rules are introduced or when the source code is changed.

3.3 Module Dependency-Restriction Rules

This section presents a few dependency-restriction rules of the CFS product line.

3.3.1 Generic to Application Modules Dependencies

As mentioned above, the product line architecture of the CFS has two major layers, namely the application layer and the core layer where the application layer is supposed to use the core layer and not the other way around. The core layer is developed for reuse in different missions. Conceptually, the application modules in the application layer need not be present in all missions. Any core module that uses application modules not only compromises the conceptual integrity of generic and specifics, but also the common look-and-feel of build rules (i.e., Makefile rules) as build targets have to be adjusted accordingly. Thus, it is important to ensure that there are no dependencies introduced from the core layer to the application layer. Otherwise, it might be difficult to build and test the core layer independently of any missions. As shown in Figure 6, the *cfs-core* layer is being used by the *cfs-apps* layer in the implementation. Of course, the modules within the application layer and the core layer are allowed to have self-dependencies, as depicted with a loop in figure 6.

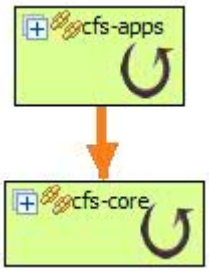

Figure 6 Implemented dependencies from the application layer to the core layer

3.3.2 Application to Application Modules Dependencies

The CFS has been architected to allow run-time plug-in of modules, even after the launch of a mission. In order to support this capability, the CFS team used the publisher-subscriber architectural style. In the implementation of this style, it is imperative that the modules in the application layer do not depend on each other directly at compile time. In other words, if two modules need to interact, they should use the services of the software bus module, defined in the core layer. Apart from the run-time adaptation capability, the CFS build process defines uniform build rules for compiling each module of the application layer into an executable.

Also, the modules are designed so that they can be tested independently of other modules. Currently, the CFS team has developed around 10 applications within the application layer. As shown in figure 7, no two application modules are communicating directly in the implementation, with an exception of a utility module (cfs_lib) which is correctly being used directly.

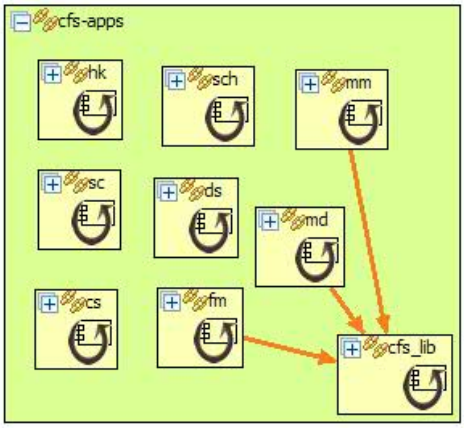

Figure 7 Dependencies among application modules

As shown in figure 8, the applications indeed communicate using the software bus only. All modules register to the software bus and exchange messages by publishing and subscribing to appropriate messages. Thus, the static structure of the publish-subscribe architecture style is in place, enabling the run-adaptation of individual (e.g., new patches or updates) modules without restarting the whole CFS.

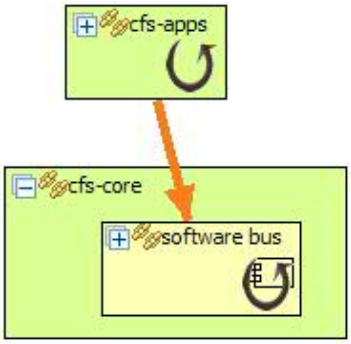

Figure 8 Dependencies on the software bus

3.3.3 Dependencies on OS and Hardware Variants

One of the goals of the CFS is to support many different operating systems (e.g., Vxworks, Unix) and hardware variants (e.g., X86, PowerPC) because they are expected to be needed by various missions. To address this need, an abstraction layer that encapsulates the underlying OS and hardware variants has been introduced. A common API for all these variants is documented in detail [6]. This API contains interfaces for using the file system, memory, and network. All the applications and core modules should be agnostic to the underlying OS and hardware. Thus, the architectural rule states that none of the modules should use the C libraries directly for accessing OS and hardware resources, and instead should go through the modules in the abstraction layer, for portability reasons. Thus, the developer must use the corresponding OSAL functions to ensure that the hardware characteristics associated with each memory address are properly taken care of. For example, attempting to write to EEPROM using the standard C function memcpy will fail. Using OS_MemCpy will succeed because the EEPROM will be configured for writing before the copy is performed. However, this rule is compromised by the core layer as it bypasses the OS abstraction layer (OSAL) and uses the C library directly (see figure 9). The SAVE tool detected that the memset and the memcpy functions are used instead of OS_Memset and OS_Memcpy defined in the OSAL layer.

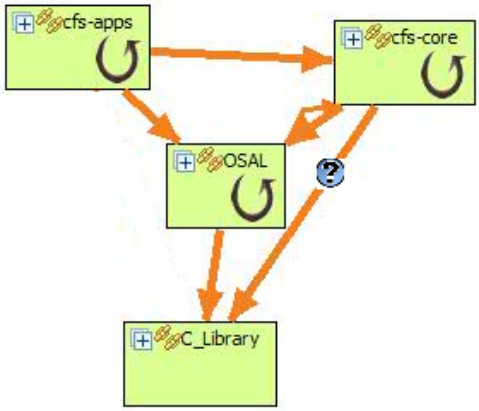

Figure 9 Bypassing the OS abstraction layer

3.4 Module Redundancy-Restriction Rules

One of the goals of the CFS product line is to minimize redundancy in the source code. The CFS team believes that implementing a product line using clones (copy-and-paste) is, in general, not a sound strategy. Clones introduce problems related to bug-fixes and maintenance. Furthermore, the source code of the CFS is also offered to some of its customers, and the presence of clones does not give a positive impression of the overall product line quality. The presence of clones also indicates that there are potential architectural design problems with respect to variability management. Implementing a product line using clones is not a good strategy, because after a few variants it is extremely hard to keep track of multiple code versions and their evolutions. In general, a product line made of clones will not be cost of effective [1] [5]. Thus, the CFS product line was verified with respect to the presence of code clones.

CloneFinder [7] , a commercial clone finder tool, is being used to locate clones. The collected clone data is at the file level. The clone finder tool outputs the detected clones as clone-groups. Each clone group contains a list of files together with line numbers of the clone. In order to analyze the collected clone data in a systematic way from an architecture perspective, the data is overlaid on the structural view of the CFS using the SAVE tool[1]. This visualization helps analyzing clones hierarchically. That is, starting from the layer-level to module-level. The analysis shows that there are no clones between the application layer and the core layer, even though both the layers are developed by the same team. The analysis of clones within the application layer showed that one function is cloned in four application modules. Similarly, one function within the core layer is cloned in two core modules. These routines could be easily moved to utility modules. Intra-module clones were also analyzed. The analysis showed that there are clones inside the memory management (MM) module of the application layer. Analysis of these clones revealed that there are only small differences between the implementations of eight, sixteen, and thirty two bit memory addresses that could be abstracted.

The OS abstraction layer (OSAL) also contains clones. This layer implements the common API [6] and consists of 150 functions for different operating systems, namely Vxworks, Rtems, Mac osx, and Linux. The clone analysis showed that 14 functions are copied in all four OS variants. Another 13 functions are copied among three OS variants. In addition, there is an overall high similarity between the Mac osX and Linux implementations of the API, supported by the evidence that around 25 functions are copied-and-pasted between these two OS variants. The Vxworks implementations differ from other OS variants primarily because the others use pthread libraries whereas Vxworks uses semaphores for multi-tasking.

Table 4 summarizes the clone measurement data. False-positives are reported as clones, but are not really clones. For example, functions for read and write operations on a file would look the same if "read" is replaced by "write". However, automatic clone finder tools have no domain knowledge and falsely report such functions as clones. Moreover, the architecture of the CFS has been designed to have a common look-and-feel among modules, and the source code is manually developed based on a template. Due to this reason, there are many false-positives reported by the clone finder tool. After manual analysis of the clone data, such false-positives were filtered out, resulting in a remaining set of true-positives. There are just a few true clones in the application and the core layer, and most of them are intra-module clones. A high number of true-positive clones are present in the OSAL. It appears that the OSAL could benefit from moving the common OS functions into a single location.

Layer	# of Clone-Group	True-Positives	False-Positives
App	48	7	41
Core	29	7	22
OSAL	69	60	9

Table 4 Summary of clone measurement

To sum up, the CFS implementation has very few clones in the application and the core layers, and the ones that do exist are under investigation by the CFS team.

3.5 Module Decomposition-Restriction Rules

As mentioned above, the CFS source code is delivered to missions, who instantiate variation points by configuring the build process and macros defined in the header files of the CFS. In order to facilitate the configuration process, the developer's guide offers rules related to decomposition of modules in the directory structure (i.e. the code architecture [11]). For example, the guide specifies in which folder the mission-specific header files and module documents have to be present, including their naming conventions.

From the CFS development team point of view, if all modules share a uniform look-and-feel, it not only helps program comprehension and evolution, but also enables developers to easily get familiar with their colleagues' implementations. Furthermore, test-suites and build scripts should also be organized in a similar way. The developer's guide provides guidelines and templates related to the structure of modules and sub-modules in the application layer. Thus, the goal of our verification is to check whether the modules of the application layer are consistent with the CFS template.

Here, a few examples of verification results are presented. The CFS template specifies that all application modules should have the directory structure as follows: The application name should be the same as the name of the sub-directory. Each application should contain a directory with the name *fsw*, which in turn contains the *src* directory, the *mission_inc* directory for configuring mission parameters, and the *platform_inc* directory for configuring platform parameters. All the application modules were verified with respect to this directory structure decomposition. The results indicate that all but one application module followed this rule (see figure 10). This deviation is marked with a red cross, meaning that the *mission_inc* directory is not present in the *sc* application module.

[1] http://www.thesavetool.com

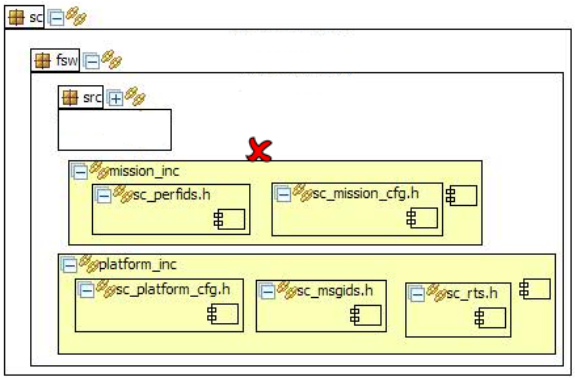

Figure 10 Directory structure decomposition - a violation: the folder mission_inc is missing

Similarly, the template explains the pattern to be followed for externally visible interfaces of each application module. It also explains how the external interface should be implemented and decomposed internally using a pseudo application module called QQ. It is expected that each module follow the structure shown in figure 11 (left), where QQ_AppMain is the only entry point to the module, and it calls QQ_AppInit, and so on as shown in figure 11.

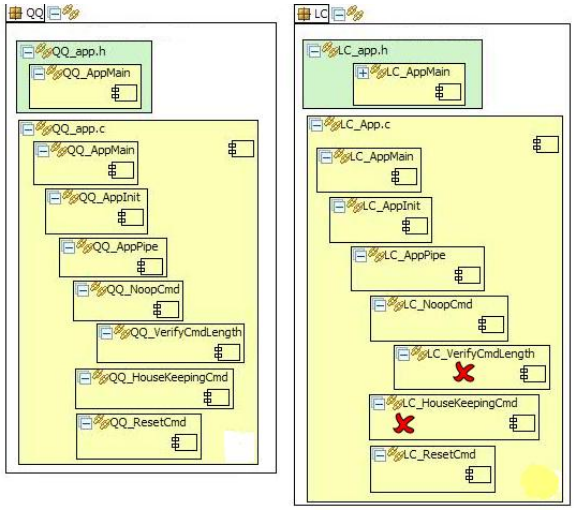

Figure 11 Left: Template. Right: Violation

All modules were verified against the QQ template, and some violations were detected (see figure 11 right). The red crosses show that the LC application module misses two routines, namely LC_VerifyCmdLength and LC_HouseKeepingCmd. Code analysis shows that the actual functionality of these routines is in fact implemented but with different routine names. Refactoring to maintain the common look-and-feel is being considered by the team.

3.6 Miscellaneous Rules and Analysis

This section summarizes the rules related to the visibility of the internals of individual modules, unused interfaces of core modules, and the usage of #ifdef/#ifndef/#if preprocessor symbols for the source code level variability management.

3.6.1 Module Internals Visibility-Restriction Rules

Since the CFS team offers the source code of the application and core modules to missions, it is very important to restrict the visibility of the internal details of individual modules. Otherwise, mission-developers might use the internals of such modules directly without using appropriate interfaces, and thus it might be difficult to later replace changed and updated CFS modules, without impacting mission-specific modules. Since the C language does not offer the concept of private and protected (as in Java), the CFS developer's guide offers coding rules. For example, one of the rules states that intra-module interfaces that are not to be used directly by mission specific applications should be declared in a header file with its name having the suffix "_priv". Moreover, no publicly visible header file should *include* a private header file; otherwise the private details are still visible indirectly to other modules for use. Similarly, none of the higher-level layer interface should expose its lower-level interface. Using the RPA and the grep tools, the *include* relations of the CFS source code were verified. The analysis showed that one of the core modules private header file was indirectly visible to external modules because a public header includes it. The CFS team is addressing this issue.

3.6.2 Core Modules Interface-Usage Analysis

The interfaces of the core modules of the CFS were developed after the commonality and variability analysis among the requirements of past missions. Thus, it is logical to expect that either all the public interfaces of the core modules are used by the application modules or the unwanted interfaces are conditionally removed (e.g., using preprocessor directives). The interface-usage analysis of the CFS core modules showed that some of the interfaces are neither used by the existing 10 application modules nor used by other core modules (see Table 5). The analysis showed that these unused core interfaces cannot be automatically removed, that is, there are no variation points to delete this unwanted functionality. The analysis also pointed out some redundancies in the interfaces of modules, implying that the service offered by an interface can also be obtained by using a combination of other interfaces. Such redundant interfaces could be easily removed to keep module interfaces minimal.

Core Module	# of Offered Interfaces	# of Unused Interfaces
Software Bus Service	30	6
Executive Service	33	2
Time Service	22	7
Table Service	15	2
File Service	6	0
Event Service	7	0

Table 5 Analysis of unused interfaces

Essentially, unused interfaces and their implementations would remain as dead code, which is considered risky in the flight software domain. In future analysis, mission-specific modules will be also included, and based on the feedback the CFS team will investigate ways to introduce variation points at the interface-level for deleting unwanted interfaces and implementations, if desired.

3.6.3 #Ifdef/Ifndef/If/elsif Complexity Analysis

The purpose of this analysis is to check how complex the implementation is with respect to the usage of conditional preprocessor directives (#ifdef, #ifndef, #if, and #elif statements) for implementing variation points. A variation point could be a binary value (e.g., Log is on or off), a numeric value, or a string value. Custom scripts were developed to measure the number of variation points per module. The measurement shows that there are around 150 variation points within the core layer and 125 in the application layer. The usage of variation points in conditional preprocessor directives were also measured using the *ifnames* tool, which emits the list of files where a preprocessor symbol is used. The histogram (see figure 12) shows that 80% of the CFS source files do not have any conditional preprocessor statements at all, excluding the guards for header files for avoiding multiple inclusions. A notable exception is one file that refers 79 variation points in a sequence of #if statements. Further analysis showed that the file validates the legal range of all variation points within a module. Overall, the use of conditional preprocessor statements has been very well under control in the CFS implementation which consists of 170 KLOC. The collected data was also used to measure cross-module interferences of variation points, which makes the source code very complex to test, comprehend, and evolve. The analysis showed that only 20 variation points out of 150 in the core layer where referred in the application layer, suggesting that the source code of application modules is using only a fraction of variation points of core modules.

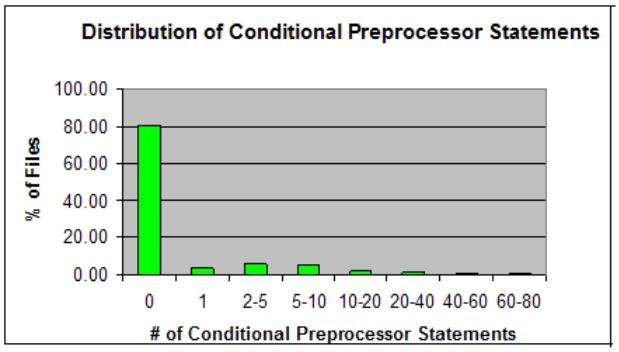

Figure 12 Conditional preprocessor directives

Also, as expected by the architectural design of the CFS application layer, there are no interference of variation points from one application module to another, and the variation points of the application layer are not referred in the core layer. This measurement indicates that there is a clear separation of concerns in the implementation.

This analysis showed that variability is implemented in the CFS using several different strategies depending on the type of variability needed. The C preprocessor is used only for configuring mission-specific parameters and removing unwanted functionalities implemented in the core and optional modules. One common API [6] and multiple implementations are developed to handle the operating system and hardware variants. The build process is designed to choose and compile the right implementation files for various operating system and hardware variants. Missions could also build and distribute the CFS modules on different CPU's; the build process can be easily configured to select the mission-specific header files, and modules are not aware of their peer modules CPU's, as all communications go through the middleware (i.e., the Software Bus module).

4 Brief Summary of Results and Lessons Learned

Table 6 presents some statistics on the number of rules checked and detected violations. The first row, for example, shows 12 different dependency-restriction rules were checked and out of that 3 rules were violated. Overall, 45 rules were checked and 14 violations were detected. Examples of detected issues were by-passing the OS abstraction layer, unexpected dependencies among modules, inter-module clones, exposing internal details of modules, redundant definitions of configuration parameters (i.e. those mentioned in #define statements) in multiple files, and a few inconsistent interface naming conventions. In addition, verification identified a module that does not use a particular interface to release memory table resources, could result in subtle performance problems.

Table 6 list the set of tools used in this process. In order to support an architecture-centric analysis, the

data collected using different tools are imported to the SAVE tool and visualized using hierarchical structural views. For example, the clone data collected from the CloneFinder tool is imported to SAVE for visualizing and analyzing clones, among from layers, modules, and sub-modules. Configuration parameters and their usage collected by the ifnames tool are imported to SAVE for visualizing and analyzing the location of variation points across layers, modules, and sub-modules.

Rule Type	# of Rules Verified	# of Rules Violated	Tools
Dependency	12	3	SAVE, RPA
Redundancy	6	3	Clonefinder, SAVE
Decomposition	5	3	RPA, SAVE
Visibility of secrets	5	1	RPA, SAVE
Variation-point Interference	3	1	ifnames, SAVE
Interface Naming Conventions	15	3	RPA

Table 6 Statistics of Rules and Violations

4.1 Lessons Learned

L1) Verifying the architectural rules helps connecting business goals, to architectural principles, and to the implementation. In this process, the teams that develop software for reuse and the teams that use the reusable software are made aware of how to develop the product line for reuse and how to reuse it in the right way.

L2) When measuring redundancy using automated clone detectors, it is worth spending effort in reviewing the detected clones. Clone detectors have no domain knowledge, thus they often falsely report similar code patterns as clones. It is easy to upset the development team with wrong clone data.

L3) Overlaying the collected source code level data, such as clones and number of variation points, onto the structural view facilitates architecture-centric analysis by showing the "big-picture" first and then details.

5 Conclusion and Future Work

This paper analyzed the CFS product line architecture by verifying that architectural rules related to the module architecture and the code architecture were indeed met in the implementation. Overall, 45 rules were checked and 14 violations were detected. It is worth noting that the CFS, as a safety-critical software product line, undergoes extensive code reviews. Nevertheless, some of the detected violations escaped the manual review process. Thus, this tool-supported verification of architecturally-significant rules complements traditional inspections by finding additional issues. The overall goal of this work is to define a method for analyzing and evaluating product line implementations from an architecture-centric perspective. This paper is the first step in that direction and the draft method will be applied to several product line implementations in the near future and will be improved based on the lessons learned. In addition, complementary analysis of rules that are outside the scope of this paper will be included. For example, rules that deal with the behavior of the system, such as task scheduling, inter-task communication, and ordering of run-time events.

References

[1] G. Böckle, P. Clements, J. D. McGregor, D. Muthig, and K. Schmid: Calculating ROI for Software Product Lines. IEEE Software 21(3): 23-31, 2004.

[2] J. Bosch. Design and use of Software architectures: adopting and evolving a product-line approach. Addison-Wesley, 2000.

[3] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. Documenting Software Architectures: Views and Beyond. SEI Series.

[4] L. Feijs, R. Krikhaar, and R. Van Ommering. A Relational Approach to Support Software Architecture Analysis. Software Practice and Experience, 28(4):371-400, 1998.

[5] D. Ganesan, D. Muthig, and K. Yoshimura. Predicting Return-on-investment for Product Line Generations. In SPLC, pages 13-22, 2006.

[6] http://opensource.gsfc.nasa.gov – The open source API for the OS and hardware abstraction layer.

[7] http://www.studio501.com/CloneFinder - The Clone finder tool.

[8] D. L. Parnas. Designing software for extension and contraction. IEEE TSE 5(2), 128-138, 1979.

[9] A. Postma and P. America. Measuring Architecting Effort. In WICSA, 2005.

[10] A. Postma. A method for module verification and its application on a large component-based system. In IST journal, 45, 171-194, 2003.

[11] D. Soni, R. L. Nord, and C. Hofmeister. Software Architecture in Industrial Applications. In ICSE, pages 196-207, 1995.

[12] C. Stoermer, L. O' Brien, and C. Verhoef. Practice Patterns for Architecture Reconstruction. In WCRE, pages 151-160, 2002.

[13] W. C. Stratton, D. E. Sibol, M. Lindvall, and P. Costa. The SAVE Tool and Process Applied at JHU/APL. In SEW, 187-193, 2007.

Acknowledgements

This work is supported by the NASA SARP program. The authors thank Charles Wildermann (head of NASA FSW Branch), and all members of the CFS team including Kequan Lu for their comments and discussions, and Sally Godrey for great support. Thanks to Dirk Muthig, Jens Knodel, Lyly Yonkwa and the SAVE team for successful collaboration in the development of the SAVE tool. Rene Krikhaar kindly provided us the RPA toolkit.