

Requirements Analysis of Variability Constraints in a Configurable Flight Software System

Chin Khor

*Computer Science Department
Iowa State University
Ames, USA
chinkhor@iastate.edu*

Robyn R. Lutz

*Computer Science Department
Iowa State University
Ames, USA
rlutz@iastate.edu*

Abstract—Variability constraints are an integral part of the requirements for a configurable system. The constraints specified in the requirements on the legal combinations of options define the space of potential valid configurations for the system-to-be. This paper reports on our experience with the variability-related requirements constraints of a flight software framework used by multiple space missions. A challenge that we saw for practitioners using the current framework, now open-sourced, is that the specifications of its variability-related requirements and constraints are dispersed across several documents, rather than being centralized in the software requirements specification. Such dispersion can contribute to misunderstandings of the side-effects of design choices, increased effort for developers, and bugs during operations. Based on our experience, we propose a new software variability model, similar to a product-line feature model, in the flight software framework. We describe the structured technique by which our model is developed, demonstrate its use, and evaluate it on a key service module of the flight software. Results show that our lightweight modeling technique helped find missing and inconsistent variability-related requirements and constraints. More generally, we suggest that a variability modeling technique such as this can be an efficient way for developers to centralize the specification and improve the analysis of dispersed variability-related requirements and constraints in other configurable systems.

Index Terms—Requirement analysis, Variability constraints, Variability requirements, Configurable system, Feature model

I. INTRODUCTION

Variability constraints are an integral part of the requirements for a configurable system [1], [2]. The constraints specified in the requirements on the combinations of options that are allowed/disallowed define the space of potential valid configurations for the system-to-be [3]. This paper reports our experience developing a variability model to specify and analyze the variability-related requirements and constraints of a NASA configurable flight software framework used by multiple space missions.

A challenge that we saw for practitioners in using the current flight software framework is that specification of its variability-related requirements and constraints are dispersed across several documents rather than being centralized in the software requirements specification. This dispersion can contribute to misunderstandings of the side-effects of design

This material is based upon work supported by the National Science Foundation under grant CCF#2211589.

choices, increased effort for developers, and bugs during operations.

Such dispersion is a persistent problem especially for configurable systems and product lines, in that information about variability-related requirements, such as options, dependencies and constraints on valid configurations, is often scattered and thus easily overlooked [4]. While much progress has been made in formal modeling and analysis of all valid configurations [5], [6], this research tends not to meet projects “where they are,” in that most projects do not have or want formal models. Consequently many projects do not find variability constraints that are missing from the requirements, are inconsistent, or are in conflict until testing or operations.

Based on our experience with the configurable flight software requirements, we propose addition of a new software artifact, a variability model, to the existing flight software framework. A variability model is a higher-level representations of options and constraints [7]. Our construction is based on a product-line feature model and is intended to be useful to future mission developers of the flight software framework. It integrates the scattered variability requirements and constraints into a coherent, centralized, variability-aware model for requirements analysis.

In this paper we describe our new, lightweight, tool-supported variability modeling technique, VarCORE, and discuss our experience applying it to a key service module of the flight software. Results reported here show that the variability model helped find missing variability-related requirements and constraints, inconsistencies among the specified variability requirements and constraints, and variability requirements and constraints not implemented in the code. We anticipate that this new modeling technique, supported by existing open-source tools, will be useful to future users of the flight software framework in both the requirements engineering and aerospace communities. More generally, we suggest that a variability model such as this can be an efficient way to centralize the specification of variability-related requirements constraints that are dispersed among documents for other configurable systems.

The contributions of the paper are three-fold:

- A proposed new variability model for the NASA configurable flight software framework that integrates, specifies

and analyzes variability-related requirements and constraints that are currently dispersed among documents

- Analysis results identifying missing and inconsistent variability requirements and constraints for repair
- A structured and partially automated technique, VarCORE, to develop and analyze the variability model for potential use in other configurable systems.

A fourth, indirect contribution of the paper is that it describes an open-sourced, configurable flight software framework that we believe merits increased use as a platform for requirements engineering innovations. To the best of our knowledge, it has not been used by the requirements engineering community, although it has been used in both architecture and testing papers [8]–[10]. Greater use of available, real-world systems such as this flight software framework benefits both requirements researchers and practitioners. Evaluation of new RE techniques on such systems can provide both real-world feedback needed to improve them and the evidence needed for their ready uptake by industry.

The rest of the paper is organized as follows. Section 2 describes the context of the NASA configurable flight software framework and the project’s variability requirements related practices. Section 3 presents the problem statement and challenges. Section 4 describes our approach in VarCORE to creating a variability model for the flight software. Section 5 presents and discusses our experience applying VarCORE to a part of the flight software and its analysis findings. Section 6 discusses lessons learned and their generalizability, and related work. Sect 7 provides concluding remarks.

II. INDUSTRIAL CONTEXT

The core Flight System (cFS) [11], [12] is a configuration-managed, software framework of flight software for spacecraft. It contains a Platform Support Package (PSP), Operating System Abstraction Layer (OSAL), Core Flight Executive (cFE) Services, cFS Applications and Tools, and supporting artifacts. The cFS applies a software product line approach to reuse across missions. Commonalities and variabilities from a rich heritage of space missions at the NASA Goddard Space Flight Center are implemented in the form of mission-independent flight software [13]. The cFS is feature-based, meaning that various combinations of features (units of functionality or configuration options) are used to meet different needs [14]–[17]. The objective is to provide high-quality, reusable core flight software that can meet the majority of basic flight requirements for a variety of spacecraft missions.

The core Flight System was open-sourced in 2015 to further the business goal of reducing time and cost for developing new flight software in spacecraft projects adopting the cFS. The second author served on the software architectural review board for the core Flight System, tasked with providing a product-line perspective, in preparation for its being open-sourced. In 2020, cFS was awarded NASA’s Software of the Year. At NASA it has been used on at least nine spacecraft projects and it is also chosen by NASA Goddard Space Flight Center (GSFC), Johnson Space Center (JSC) and Johns

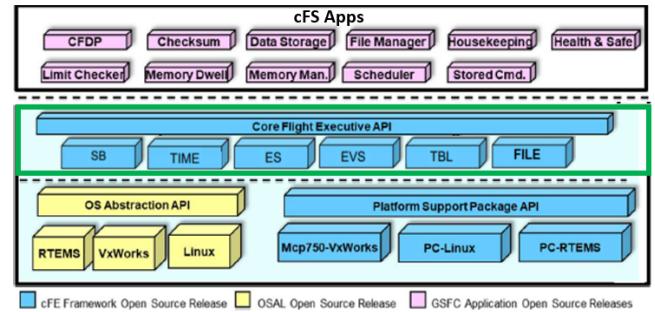


Fig. 1: cFS Software Architecture and Organization. Excerpt from *Core Flight System (cFS) Training*, p. 9 [20].

Hopkins University Applied Physics Lab (APL) for all future embedded flight software projects [11]. As part of the Lunar Gateway program, work is also underway to certify cFS as suitable for human-rated vehicles [18]. According to the team lead for certification, “We work on maybe two or three missions a year, but outside of NASA, people are trying it out, finding new ways to use it, and making suggestions for improvement” [13]. Examples include the CubeSat nanosatellites and small spacecraft [12]. Educationally, OpenSatKit (OSK) adopts cFS to provide a free flight software system platform for aerospace and STEM education [19].

A. Architectural Design for Variability Support

The cFS was designed with reusability and variability in mind. Figure 1 shows the layered modular architecture of the cFS. At the bottom layer, the cFS supports multiple hardware platforms (including i686/x86 based PC, ARM-based PC, powerPC-based PC and MCP750) and operating systems (including Linux, VxWorks and RTEMS) via abstractions by OSAL and PSP. It thus hides the implementation details of hardware and operating systems from the applications and executive.

The core Flight Executive (cFE) provides core services. At the top layer, the cFS defines standardized core Application Programmer Interfaces (APIs), which are the only interfaces for applications to access the cFE core services. Although the cFE core services are common across most flight system projects, the cFE does provide configurable parameters to determine variations of the core services at compilation. To change core services’ configuration, the entire cFE must be recompiled and restarted with a system reset.

The cFS is designed to support plug-and-play applications using these standardized APIs. This means that applications can be loaded and removed at runtime. Applications define mission functionality with the support of the cFE. The cFS architecture ensures that different combinations of applications can be run independently and concurrently to fulfill mission requirements.

B. Strategies for Variability Management

There are three strategies for managing software product variability in the cFS. First, the cFS uses its architecture design to manage mission-specific application variability. Each

application is an independent entity. To establish connection with the cFE, each application is required to register and subscribe for services' access to cFE. Inter-application interaction is only supported via cFE, i.e. direct interaction is constrained. Ganesan et al. [8] analyzed the cFS product line architecture and verified that there is no inter-dependency and direct interaction among applications, complying with architectural design for the cFS application layer.

Second, the cFS uses its build system to manage hardware variants, OS variants, and module and test configurability. The cFS build system checks for configurable parameters during compilation and builds the flight software product accordingly. Thus, applications and cFE source code can be run on different hardware and OS configurations via the support of the abstraction layer and defined APIs.

Third, the cFS uses conditional compilation C pre-processor (CPP) directives (such as `#if`, `#ifdef`, `#elif`, `#ifndef`) to compile the code based on the setting defined by configurable parameters. There are 139 configurable parameters [12] defined in the cFE 6.7.0 release that are handled by conditional compilation. We found that only 15 of the configurable parameters represent variant features for configuring the system functional behavior. The rest of the configurable parameters are mainly defined to represent system (i.e., platform and mission) properties such as memory size, message id, event type, etc. The cFE uses CPP directives to verify the validity of these system properties before system configuration.

III. PROBLEM STATEMENT: VARIABILITY-AWARE REQUIREMENTS CONSTRAINTS

The cFS describes their requirements strategy [21] as incorporating requirements engineering best practices to manage software variability. The requirements are parameterized to support various missions. Fig. 2 shows the software engineering V-Model for the cFS-based project life cycle. On the left, the requirements are refined from high-level system requirements into lower-level detailed functional requirements for flight software development. On the right, corresponding tests are created to verify/validate the requirements at each

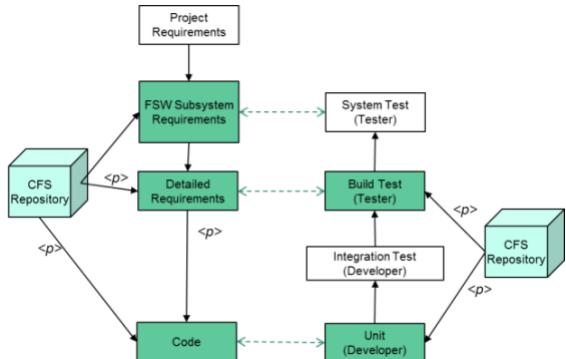


Fig. 2: V-Model of cFS-based Software Development Lifecycle. Reprinted from [12]. The shaded components are cFS artifacts and `<p>` notation indicates a parameterized artifact.

level. In addition, the functional requirements are written using recommended practices to assure quality [22]. Any change in requirements requires review and approval before becoming effective.

Nevertheless, there remains a risk that variability constraints may be overlooked during requirements refinement, especially if selections of options are delayed until design or compilation phases. The cFS uses pre-processor directives to verify and assure the consistency of variability requirements and constraints at build time. However, we observed that the constraint verification is done independently without explicit traceability to the variability requirements. Moreover, it is known that the gap between requirements specification and implementation tends to grow [23] if requirements updates are not performed in a timely manner.

Another challenge that we observed with the current framework is that the variability-related requirements are not specified in a single requirements specification document. Instead they are dispersed across several documents. Some constraints are mentioned but not described, or are described but not specified as requirements. Such dispersion can complicate the requirements analysis and thus increase the effort for variability design, implementation and testing.

The cFE Flight Executive Software Requirements Specification is relatively short (30 pages plus appendices). For example, it contains only 32 requirements for the six cFE services. The other requirements document, the Functional Requirements table, has 398 brief (one-line) "shall" textual requirements. It does not document the configuration options available nor the intra-cFE variability constraints that the developers of a new product need to know. On the other hand, the cFE user guide is very long (1,056 pages). It specifies requirements for the configuration options and the variability constraints. However, its length renders it difficult to navigate and use.

The dispersion of variability-related requirements and constraints among documents creates a Goldilocks problem [24], named after the children's story in which Goldilocks finds the Papa Bear's chair too big and the Mama Bear's chair too small, but the Baby Bear's chair is "just right" for her. The analogy refers to the challenge that developers of a new system using the cFE face in currently not having a central "just right" sized specification.

Towards making it easier for developers to understand the variability features and constraints on the feature combinations, we built a new, visual variability model that captures this requirements information in a standardized and familiar graphical form. We do not claim that it is Goldilocks-optimal; however, show that it facilitates discovery of missing and inconsistent variability-related requirements and constraints.

IV. APPROACH

In this section we describe our VarCORE technique and the new software variability model it produces, proposed for inclusion in the flight software framework. Fig. 3 gives an overview of VarCORE and the artifacts it uses and provides for

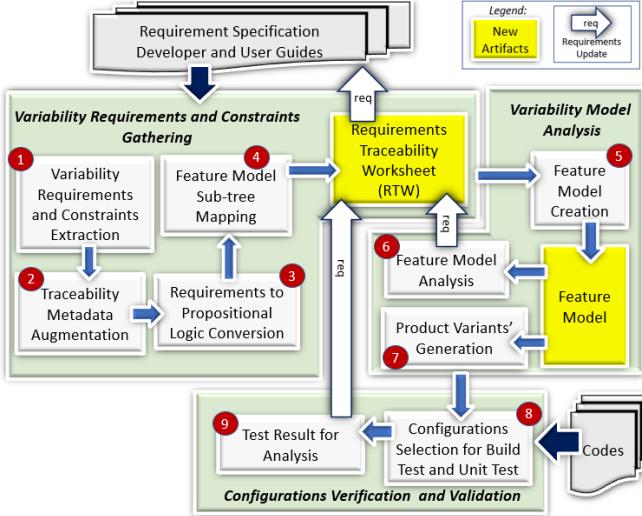


Fig. 3: VarCORE Overview. The new artifacts produced by the variability-related requirements and constraint analyses are highlighted in yellow.

variability-related requirements and constraints analysis in a configurable flight software. The circled numbers in the figure map to the elements in the description below.

There are three components in the VarCORE process: (1) Variability Requirements and Constraints Gathering, (2) Variability Model Analysis, and (3) Configurations Verification & Validation.

A. Variability-Related Requirements and Constraints Gathering

The primary input to VarCORE is the software requirement documents. For large-scale software projects there are typically more than one requirements document, starting from system requirements and decomposing them into subsystems, modules and units requirements. Often these are produced by different engineering teams at different levels of refinement. This can create a challenge to maintaining consistency and traceability among all requirement documents.

In our approach, we propose a lightweight process to gather the scattered variability requirements and constraints from diverse requirement documents into a centralized Requirements Traceability Worksheet (RTW), illustrated in Table I. Capturing in the RTW the dispersed information needed to construct the feature model enables the step-by-step construction of a feature model that is consistent with the documented variability-related requirements and constraints.

First, the variability requirements and constraints are extracted from the requirement documents and entered into

TABLE I: Requirements Traceability Worksheet (RTW).

Requirement Traceable Worksheet (RTW)							
Doc Section Number	Doc Section Header	Variability-Related Requirements or Constraints	Requirement ID	Abstract Features	Concrete Features	Conversion to Propositional Logic	Feature Model Sub-Tree Mapping

the RTW, one requirement/constraint per entry. Then, we identify the relevant terms such as the variant features and/or the feature constraints from each requirement/constraint, and name them as concrete features which are key elements of a feature model. In implementation, these terms are generally the configurable parameters that manage variability or establish constraints in the code. As an example, “*cFE is configured to function as a time server or time client*” is an identified variability requirement. Clearly *time server* and *time client* are two configuration variants of the *cFE* function. Accordingly, we define *time server* and *time client* as concrete features, and group them under the *cFE function*. Usually concrete features are terminal nodes of a feature model. However, we chose to allow concrete features in non-terminal nodes to keep the feature tree smaller. Our feature tree still can be transformed into a GenVoca feature tree [1]; that transformation adds nodes, so we avoid it here.

Second, we assign a unique requirement ID as an identifier for each entry in the RTW. Then we augment the traceable metadata when naming the abstract feature [1], [25], [26]. In the context of a feature model, the abstract features are non-terminal nodes used to structure the feature diagram by grouping the related concrete features (identified in the previous step), so the name of an abstract feature should be representative for grouping concrete features. Our naming convention is to augment the abstract feature name (e.g. “name”_“id”) with traceable metadata that is extracted along with the requirement/constraint. The type of metadata is varied depending on the requirement structure. If the requirements/constraints are specified in the structured requirement document, the metadata can be the requirement ID and the heading description for each requirement/constraint, e.g., Time_Function_T1001. On the other hand, if the requirements/constraints are extracted from a free-form textual requirement document, the metadata can be document section number and heading, e.g., Time_Server_Client_1_22_4_5. Although abstract features are essential elements for a feature model, they do not contribute to product variant generation as they do not appear in the code.

Third, after identifying the abstract and concrete features for each variability requirement/constraint, VarCORE translates the textual variability requirements and constraints into propositional logic expressions [27] using the propositional logic forms in Table II. For example, “*cFE is configured to function as a time server or time client*” will be translated to

$$\text{Time_Function_T1001} \iff$$

$$(\text{time server} \wedge \neg \text{time client}) \vee (\neg \text{time server} \wedge \text{time client})$$

This translation is done manually, and we discuss the limitations of NLP techniques that drove our decision not to automate this in related work in Section VI-C. The translation to propositional logic form serves two purposes. One is to help verify the correctness of the variability requirement or constraint. A requirement or constraint statement in natural language can be ambiguous and interpreted differently by different developers. Translating it to propositional logic which has only one mathematical interpretation helps verify the correctness of the requirement or constraint.

TABLE II: Rule Definition for Mapping Propositional Formulas to Feature Model Sub-trees.

Rules	Propositional Formula	Semantics (Parent-Child Relationship)	Feature Model Sub-tree
R1	root	Root (mandatory)	
R2	$A \Leftrightarrow B$	Select Mandatory Child	
R3	$C \rightarrow A$	Select Optional Child	
R4	$A \Leftrightarrow (B \wedge \neg C) \vee (\neg B \wedge C)$	Select Only 1 Child	
R5	$A \Leftrightarrow (B \vee C)$	Select At Least 1 Child	
R6	$(A \Leftrightarrow (B_1 \wedge B_2)) \wedge ((C_1 \vee C_2) \rightarrow A)$	Select All Mandatory and Any Optional Child	
R7	$C \rightarrow (Y \vee R \dots) \wedge (\dots \rightarrow (Y \wedge \neg R \dots) \vee (\dots))$	Cross Tree Constraints	

Fourth, once in propositional logic form, the relationship between abstract and concrete features can be determined using the defined mapping rules [1] based on Feature Oriented Domain Analysis (FODA) notation [28] (see Table II). Features are units of functionality, i.e., functional requirements [14]–[16]. These rules convert abstract features, concrete features and their relationships to feature model sub-trees. For instance, the propositional logic expression “*cFE is configured to function as a time server or time client*” is mapped to rule R4, which states that a Time_Function_T1001 (abstract feature) can select only one child (concrete feature), either *time server* or *time client*. By following the rules given in Table II, we can systematically convert all variability requirements and constraints in the RTW to their associated feature model sub-trees. Table IV shows a sample of the RTW generated for the Time service module of the cFE.

The RTW is composed of entries for variability requirements and variability constraints. Two variability requirements are in a hierarchical relationship if the feature of one variability requirement is the parent node of the feature (i.e., child node) of another variability requirement. Thus, a hierarchical relationship between two variability requirements is also a parent-child relationship. A variability constraint is represented by another type of relationship, the cross-tree constraint, which is an implication relationship (i.e., if-else constraint) between two or more concrete features across variability requirements.

Evaluation to assure consistency among variability requirements and constraints is not a trivial task. VarCORE creates a graphical variability model (using feature modeling) to aid the variability requirements and constraint analysis (see next section). Additionally, to handle requirement changes, entries can readily be added, modified, or deleted in the RTW at any time to create updated or experimental variability models for comparison, analysis, or troubleshooting.

B. Variability Model Analysis

Once the first new artifact, the RTW, is constructed, we use it to generate a second new artifact, a feature model that represents the variability requirements and constraints in a graphical view. A feature model is a standard type of variability model, with table-based models being another common type. A feature model is a feature diagram, described below, together with the constraints on those features [1], [28].

In VarCORE we employ the open-source tool FeatureIDE, which has been widely used for feature modeling [29] to construct the feature model (refer to the fifth step in Fig. 3). We formulate each feature model sub-tree (from RTW) in the Extensible Markup Language (XML) format recognized by FeatureIDE. Within a sub-tree, the hierarchical relationship between abstract and concrete features must be maintained. The XML representations of the sub-trees then are concatenated and integrated into an XML file for the feature model. Fig. 4 shows the process of concatenating variability requirement and constraint sub-trees into the XML template recognized by FeatureIDE. The order of concatenation for sibling sub-trees and constraints does not impact the feature model computation, but does decide features’ and constraints’ relative display position in the feature diagram. We developed a program¹ to automatically construct a feature model (XML format) from the structured RTW entries. Once the feature model in XML format is created, we input it to FeatureIDE to generate a graphical representation of the feature model automatically (see Fig. 7 for the final feature model generated for the cFE Time service module).

We use FeatureIDE’s built-in capabilities to analyze the model’s validity, including checks for void-model detection (i.e., feature model without any valid configuration), dead-feature detection (i.e., a feature that is not part of any valid configuration), false-optional feature detection (i.e., optional feature but present in all products), constraint redundancy, and conflict of constraints. During the feature model analysis, FeatureIDE flags any occurrence of these errors in the feature diagram and reports them in its statistical report. Using our RTW, we then can trace any reported inconsistency or error both back to the corresponding variability requirement or constraint, and from there to its source in a requirement document. Fig. 5 shows the traceability among the requirements documents, the RTW and the feature model.

After completion of the feature model analysis, FeatureIDE can generate all valid configurations (or product variants) encoded in the feature model. These valid configurations are then exported to a repository for our subsequent use.

C. Verifying and Validating Configurations

The feature model undergoes static analysis to check the consistency among the variability requirements and constraints. However, the analysis does not detect variability bugs in the implementation. To detect inconsistencies between the specification of the variability requirements and constraints in

¹<https://github.com/chinkhor/VarCORE/tree/main/scripts/RTW2FeatureModel>

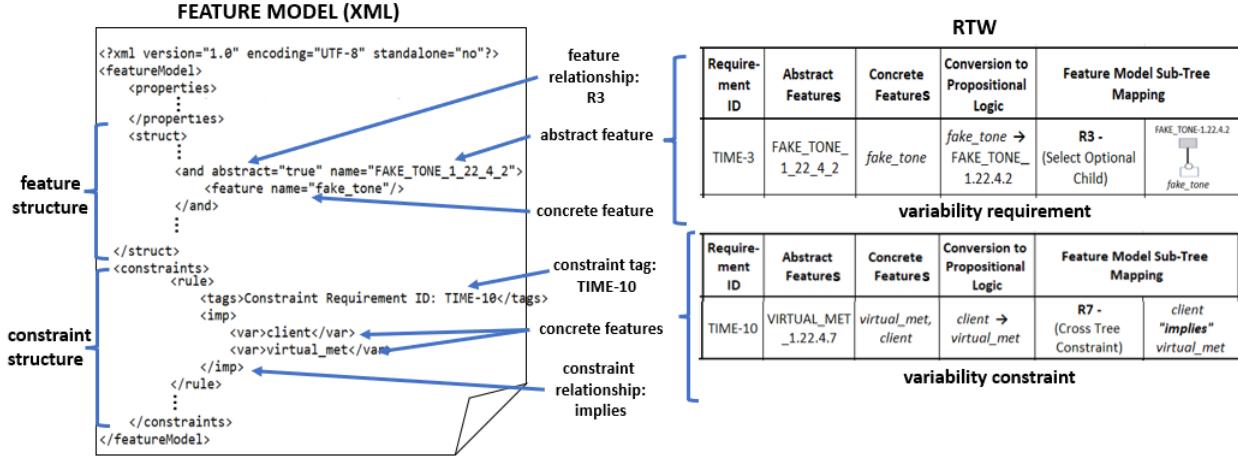


Fig. 4: Concatenating Feature Model Sub-trees using XML Template for Feature Model Construction.

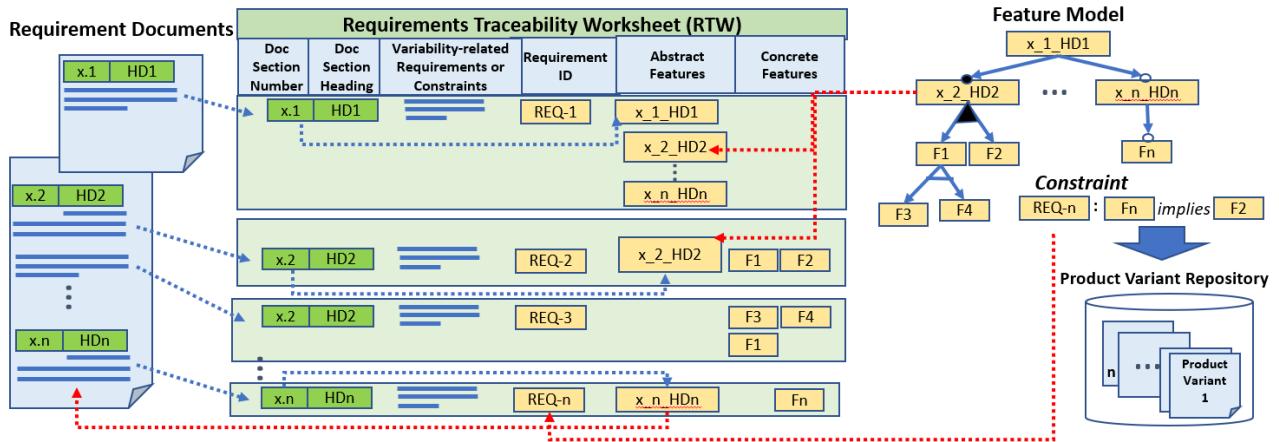


Fig. 5: Traceability among Requirement Documents, RTW and Feature Model.

the feature model and the source code, we use the generated product variants from our feature model as inputs to the cFS build system and unit test framework for variability bug detection. Each product variant represents a valid configuration that satisfies all variability requirements and constraints specified in the RTW. Verifying these product variants via build tests and unit tests helps assure consistency between the variability requirements/constraints and the code that should satisfy them, while inconsistencies could indicate flaws in the requirements.

The product variant files generated by FeatureIDE are in XML format. They thus must be translated to configuration files that are recognized by the build system. To achieve this, we parse and convert the concrete features in XML to the corresponding variable names in the source code. This step uses a pre-defined mapping table between concrete feature names and code variable names. The build system then modifies the configuration setting as specified by the product variant configuration file to start the build test and unit test. If the configuration space is small, all product variants can be selected for testing. Otherwise, random or targeted sampling of product variants for testing can be done.

Finally, the test results for each product variant is gathered for analysis. Any build failure or unit test error is an indication of potential inconsistency between the implementation and the variability requirements/constraints. Once the variability bug is identified, the RTW can be used to trace the source of variability requirements and constraints which are related to the inconsistency for further analysis.

V. APPLICATION AND FINDINGS

In this section we describe our application of VarCORE on a portion of the cFS flight software. We also present our findings regarding the following three research questions:

- RQ1: Are there missing variability-related requirements and/or constraints?
- RQ2: Are there inconsistencies among the specified variability requirements and/or constraints?
- RQ3: Are there variability requirements and/or constraints not implemented in the code?

A. Application

We chose to evaluate our VarCORE technique on the cFE Time service module (TIME) since 14 of the 15 variant feature

TABLE III: TIME Configuration Variability and its Functionality.

Variability	Functionality
<i>server</i>	Time operation in Server mode
<i>client</i>	Time operation in Client mode
<i>big_endian</i>	Force tone message in big endian order
<i>virtual_met</i>	Configure as virtual MET if there is no local hardware MET. <i>virtual_met = false</i> indicates local hardware MET is enabled.
<i>source</i>	Source of time data is external
<i>source_met</i>	Type of external time data source is MET
<i>source_gps</i>	Type of external time data source is GPS
<i>source_time</i>	Type of external time data source is spacecraft time
<i>signal</i>	Support primary and redundant tone signal selection
<i>tai</i>	Default time format is International Atomic Time
<i>utc</i>	Default time format is Coordinated Universal Time
<i>fake_tone</i>	Enable fake tone signal generation in the absence of real hardware signal
<i>at_tone_was</i>	Tone signal arrives before "time at the tone" data
<i>at_tone_will_be</i>	Tone signal arrives after "time at the tone" data

configurable parameters that define cFE system behavior are used for Time configuration. TIME is the core service that provides spacecraft time correlation, distribution, and synchronization services. These encompass multiple variabilities, including the time operational mode (server or client mode), time format, source of time, scheme for tone signal, etc. Table III shows all 14 TIME configuration variabilities and their corresponding functionalities. The configuration of these variabilities is defined at build-time and dictate the TIME system behavior at run-time.

Initially we selected the cFE Software Requirement Specification (cFE requirements.docx) and the cFE Functional Requirements specification (cFE FunctionalRequirements.csv) as the sources from which to gather the variability-related requirements and constraints for TIME. However, we found that only six variability requirements are described in these two requirements documents. We then discovered that, instead, the cFE User's Guide (which is the reference for application, tool and test development) describes the TIME variability requirements and constraints in detail. We thus used it as one of the primary input documents for our VarCORE application and evaluation.

B. Findings

Table IV shows an excerpt of the first generated artifact produced with VarCORE, the RTW (Requirements Traceability Worksheet) described in Section IV-A. Following the process described there, 10 variability requirements and 7 variability constraints were identified for TIME configurations and extracted from the cFE User's Guide. Then, 9 abstract features and 15 concrete features were derived from the 17 variability requirements and constraints.

With TIME's variability-related requirements and constraints now gathered into the structured RTW, we translated each requirement or constraint into its propositional logic formula. In parallel, we reviewed all the formulas to verify the correctness and consistency of the variability requirements and constraints for the TIME configuration.

Interestingly, in the review process we discovered that one cross-tree constraint could never be satisfied. TIME-7 (see

Table IV) could not be satisfied as there was no definition for processor count, neither in TIME, nor in PSP or OSAL (we manually checked PSP and OSAL later). TIME-7 thus was flagged as a unsatisfiable variability constraint and highlighted in red on the RTW.

After verification of the variability requirements and constraints was completed, we created the feature model for TIME using the FeatureIDE tool (described in Sect. IV-B) using the satisfied variability requirements and constraints but without TIME-7. The graphical feature model is the second artifact produced by VarCORE, for use in the feature model analysis.

Fig. 6 shows the *initial* constructed feature model, including the cross-tree constraints, for TIME that was generated by the FeatureIDE tool. However, FeatureIDE reported the existence of a false-optimal feature in the initial feature model. It showed that this was caused by the constraints specified by TIME-10, TIME-11 and TIME-16, as shown in the figure.

Further investigation by the authors indicated that the culprit was TIME-11, as this constraint would prohibit the Time Server from using the local hardware's MET (Mission Elapsed Time). The variability constraint specified in TIME-11 was thus judged to be erroneous and was highlighted in red in the RTW for attention by the developers. Using the traceability provided by the feature model and RTW, we could readily identify the source of this requirement specification, so that it could be repaired. Figs. 7 and 8 show the updated result, a valid feature model, after the variability constraint TIME-11 was removed to resolve the inconsistency.

After completion of the model analysis, we generated all 112 product variants for verification and validation using cFS's build system and unit test framework. Each product variant represents a valid configuration supported by the feature model. Our goal is to assure that the cFE does fulfill the variability requirements and constraints specified in the RTW. We transformed all 112 product variants (in XML files) to configuration files recognized by the cFS build system using the mapping table shown in Table V. The list of C pre-processor (CPP) directives for TIME was retrieved using the Linux ifnames tool, and the mapping was straight forward as there were common keywords between the concrete features and the CPP variable names.

Table VI shows that 76 out of the 112 valid configurations failed the build tests for the unit tests, where 64 of the failures were the same as the system's failed build tests. After investigation, all the failures of both system and unit test builds were found to be caused by three variability bugs. TIME attempts to call *unimplemented abstract functions* (*OS_SelectTone*, *OS_SetLocalMET* or *OS_GetLocalMET*) when either a selection for the tone signal (*cFE_PLATFORM_TIME_CFG_SIGNAL=true*) or for the local hardware MET (*cFE_PLATFORM_TIME_CFG_VIRTUAL=false*) is configured. Additionally, there was a unit test coding bug (TIME-6 (*cFE_PLATFORM_TIME_CFG_CLIENT=true*)) that was found to occur when the Time Client is configured.

These build failures indicate that the code does not satisfy the following variability requirements (refer to Table IV for

TABLE IV: Excerpts from Requirements Traceability Worksheet (RTW) for cFE Time Configuration. Variability requirements and constraints that are unsatisfiable or conflicted with others are highlighted in red. The full RTW is available at <https://github.com/chinkhor/VarCORE>.

Requirement Traceable Worksheet (RTW)							
Doc Section Number	Doc Section Header	Variability-Related Requirements or Constraints	Requirement ID	Abstract Features	Concrete Features	Conversion to Propositional Logic	Feature Model Sub-Tree Mapping
1.22.4.2	Enabling Fake Tone Signal	Enable the fake tone only for hardware configurations that do not provide a tone signal.	TIME-4	FAKE_TONE_1_22_4_2	fake_tone virtual_met source	$fake_tone \rightarrow (virtual_met \wedge \neg source)$	R7 - (Cross Tree Constraint)
1.22.4.5	Specifying Time Server/Client	Enable one, and only one, of the following definitions in the platform configuration header file: #define CFE_PLATFORM_TIME_CFG_SERVER TRUE #define CFE_PLATFORM_TIME_CFG_CLIENT FALSE or #define CFE_PLATFORM_TIME_CFG_SERVER FALSE #define CFE_PLATFORM_TIME_CFG_CLIENT TRUE	TIME-6	TIME_SERVER_CLIENT_1_22_4_5	server client	$SERVER_CLIENT_1_22_4_5 \Leftrightarrow (server \wedge \neg client) \vee (\neg server \wedge client)$	R4 - (Select Only One Child)
1.22.4.5	Specifying Time Server/Client	If the target system has only one processor running the cFE, then TIME must be configured as a server.	TIME-7	TIME_SERVER_CLIENT_1_22_4_5	processor_count server	$(processor_count==1) \rightarrow server$	R7 - (Cross Tree Constraint)
1.22.4.7	Virtual MET	If the MET is not local then TIME must be configured as using a virtual MET	TIME-9	VIRTUAL_MET_1_22_4_7	virtual_met	$virtual_met \rightarrow VIRTUAL_MET_1_22_4_7$	R3 - (Select Optional Child)
1.22.4.7	Virtual MET	Therefore, all TIME clients must be configured as using a virtual MET.	TIME-10	VIRTUAL_MET_1_22_4_7	virtual_met client	$client \rightarrow virtual_met$	R7 - (Cross Tree Constraint)
1.22.4.7	Virtual MET	TIME servers must be configured as using a virtual MET	TIME-11	VIRTUAL_MET_1_22_4_7	virtual_met server	$server \rightarrow virtual_met$	R7 - (Cross Tree Constraint)
1.22.4.9	Specifying Time Signal	Setting the following configuration definition to TRUE will result in enabling a TIME command to select the active tone signal. #define CFE_PLATFORM_TIME_CFG_SIGNAL TRUE	TIME-15	TIME_SIGNAL_1_22_4_9	signal	$signal \rightarrow TIME_SIGNAL_1_22_4_9$	R3 - (Select Optional Child)
Scattered Constraints		Time Server cannot be defined as having both a h/w MET and an external time source.	TIME-16	none	virtual_met source	$source \rightarrow virtual_met$	R7 - (Cross Tree Constraint)

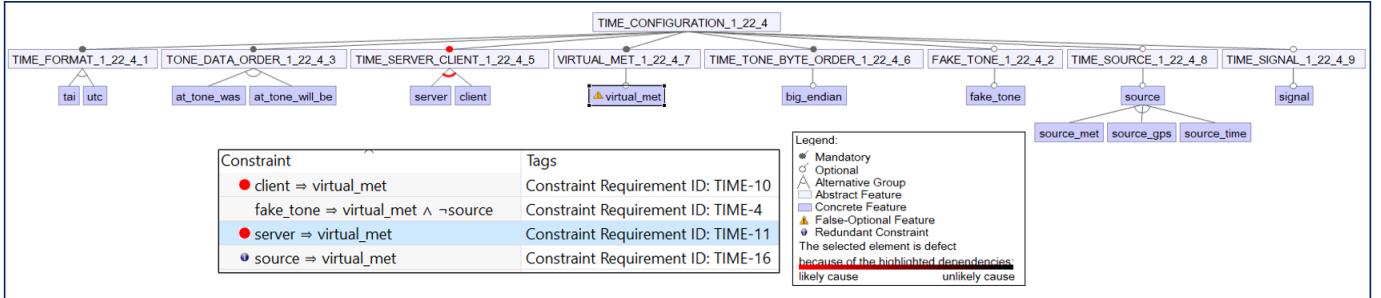


Fig. 6: Initial Constructed Feature Model Showing Detection of Inconsistent Constraints.

TABLE V: Mapping between Concrete Features and Code.

Concrete Features in Feature Model	C Pre-processor Directives in CFE Time Service Module (Codes)
server	CFE_PLATFORM_TIME_CFG_SERVER
client	CFE_PLATFORM_TIME_CFG_CLIENT
big_endian	CFE_PLATFORM_TIME_CFG_BIGENDIAN
virtual_met	CFE_PLATFORM_TIME_CFG_VIRTUAL
source	CFE_PLATFORM_TIME_CFG_SOURCE
source_met	CFE_PLATFORM_TIME_CFG_SRC_MET
source_gps	CFE_PLATFORM_TIME_CFG_SRC_GPS
source_time	CFE_PLATFORM_TIME_CFG_SRC_TIME
signal	CFE_PLATFORM_TIME_CFG_SIGNAL
tai	CFE_MISSION_TIME_CFG_DEFAULT_TAI
utc	CFE_MISSION_TIME_CFG_DEFAULT_UTC
fake_tone	CFE_MISSION_TIME_CFG_FAKE_TONE
at_tone_was	CFE_MISSION_TIME_AT_TONE_WAS
at_tone_will_be	CFE_MISSION_TIME_AT_TONE_WILL_BE

more information):

- TIME-9 (CFE_PLATFORM_TIME_CFG_VIRTUAL=false)
- TIME-15 (CFE_PLATFORM_TIME_CFG_SIGNAL=true)
- TIME-6 (CFE_PLATFORM_TIME_CFG_CLIENT=true) in

TABLE VI: TIME Build Test and Unit Test Results.

Build Test Component	Results		Total Configurations
	Pass	Fail	
cFS Time Service	48	64	112
cFS Time Unit Test	36	76	112

unit test code only.

C. Research Questions

We now discuss what our findings from application of the VarCORE on the cFE indicate in terms of the research questions.

RQ1: Are there missing variability-related requirements and/or constraints?

Several variability requirements and constraints are missing from both the cFE Software Requirement Specification (cFE requirements.docx) and the cFE Functional Requirements document (cFE FunctionalRequirements.csv). During variability requirements and constraints gathering, we found that together they only describe 6 of the variability requirements for the

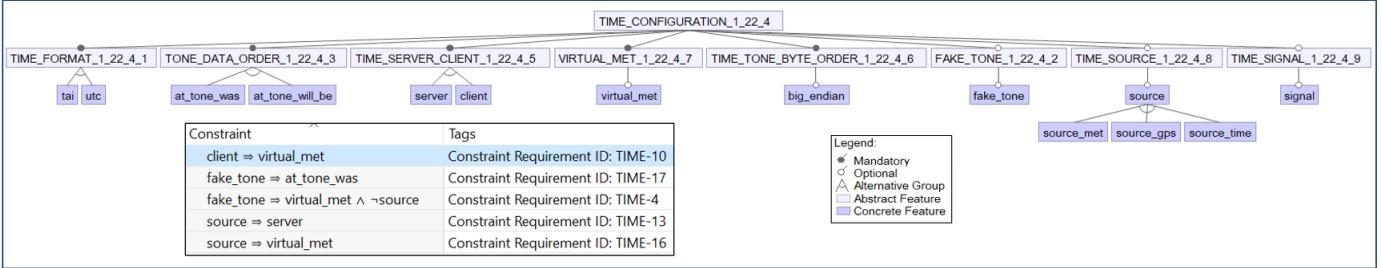


Fig. 7: Final Feature Model for Time Service Configuration. The full feature model is at <https://github.com/chinkhor/VarCORE>.

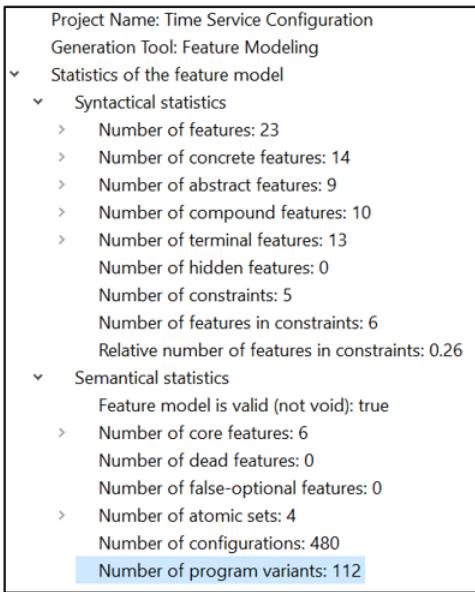


Fig. 8: Feature Model Analysis Statistics by FeatureIDE

TIME module compared to 17 (10 variability requirements and 7 constraints) that we extracted from the cFE User's Guide. There is currently no centralized requirement document from which all behavioral requirements and constraints could be identified and understood.

RQ2: Are there inconsistencies among the specified variability requirements and/or constraints?

Review of the RTW identified a variability requirement inconsistency during the review process. TIME-7 (see Table IV) cannot be satisfied as there is no definition for processor count in TIME nor imported to TIME. We also found through the feature model analysis that the constraint TIME-11 conflicted with TIME-10 and TIME-16. These issues require requirements updates to address the inconsistencies.

RQ3: Are there variability requirements and/or constraints not implemented in the code?

We investigated this question by testing all 112 valid configurations (product variants) using cFS's existing build system. The results show that TIME-9 and TIME-15 are not implemented in the code. Moreover, there is a variability bug found in unit test for supporting TIME-6. All three of these variability bugs are known open issues tracked by cFS's Bug ID #109 (<https://github.com/nasa/cFE/issues/109>).

We analyzed the history of resolved bugs for TIME and found a variability bug #2072 (<https://github.com/nasa/cFE/issues/2072>) which occurred only for a specific TIME configuration, one of the valid 112 product variants generated by VarCORE. We reproduced the bug (temporarily undoing the fix) when running the build test using the specific product variant as TIME configuration and confirmed that the variability requirement was correct but not implemented correctly in the code. This resolved bug could have been detected earlier if VarCORE had been integrated into the cFS build system.

During code review for answering RQ3, we also found that the specified constraint described in the RTW for TIME-4 is inconsistent with the code implementation. TIME-4 specifies that the issuance of a fake tone shall be enabled only for hardware configurations that do not provide a tone signal. Since both local hardware MET and external source MET can provide a tone signal, the fake tone should be disabled in these two configurations. However, the implementation of this variability allows a fake tone when the local hardware MET is enabled. The implemented constraint is inconsistent with the specified constraint.

VI. DISCUSSION

A. Lessons Learned

We report four lessons learned from our experience with the flight software that may be useful on other industrial projects.

- Information about variability-related requirements and constraints may be dispersed among documents, not all of which are labeled as requirements. This makes it more time-consuming to identify and understand optional requirements and constraints on the design space. It also makes it easier to miss needed requirements or to inadvertently violate constraints by introducing conflicting requirements.
- Variability-related requirements and constraints are easily overlooked and need special attention in mission-critical software. The flight software framework uses many best practices for variability management, as described in Sect. II, and is well maintained by experienced developers. For a new project with developers initially unfamiliar with the flight software framework, there remain obstacles to navigating the maze of information [4] to determine their project requirements. Our findings

(Section V) suggest that special attention to variability-related issues can simplify future developers' tasks.

- Lightweight variability modeling, as in VarCORE, appears to be effective in finding errors. We identify two reasons for this. (1) It focuses special attention on the variability. (2) It centralizes the dispersed requirements in a structured format (the RTW) that enables automated creation and analysis of a standard feature model.
- A graphical view of the variability requirements and constraints makes it easier to see and understand inconsistencies. VarCORE uses a well-known tool (FeatureIDE) to display the feature model in a tree-based, graphical display annotated with any constraint violations, and to generate all valid configurations in support of build and unit testing.

An interview-based study 2021 by Schmid et al. noted that “industry still struggles to deal with a high number of variants of their systems systematically. The underlying issue, i.e., that knowledge about variability is often only tacit, available from the heads of the developers only, has not disappeared” [30]. Similarly, Kruger et al. reported in a 2019 study that the availability of lightweight traceability of features to source code immediately benefited both developers and maintainers [31]. We sought in our work to improve the state of practice by making the knowledge about constraints on valid feature combinations less tacit and more obvious to the developers of a new flight software system using the cFS framework.

B. Threats to Validity

An internal threat to validity is that, while verification that our feature model is sound uses the static analysis performed by the FeatureIDE tool to confirm that it produces only valid configurations, the variability requirements or constraints themselves may be incorrect. VarCORE can assist to some extent in surfacing such errors since it checks that all the configurations identified as valid by FeatureIDE can be compiled and built for unit testing. Another threat to internal validity is that several steps in VarCORE are manual (requirement/constraint extraction, propositional logic conversion) and require domain expertise, with results dependent on the accuracy of input provided by developers. We can automate more of the flow to lessen this dependency; however, the current worksheet-to-model approach has the benefit of being familiar with a low bar to adoption. Our selection of one project, the NASA flight software, for application may affect the external validity. However, we chose it due to our experience with it, and it has a sizeable community that has used it on a variety of aerospace projects.

C. Related Work

Variability models are well-studied and take multiple forms including feature models [28], tabular configurability models [4], and formal models [32], created either from requirements or reverse engineered from code.

Automating requirements and feature extraction using NLP remains a challenging task and requires domain expertise to

fine tune the tools and verify the results. At RE’22 Rajbhoj et al. described successful extraction of traceability information from requirements specifications that are highly structured, which was not the case here [33].

Our approach is perhaps most similar to that described by Acher et al., who also proposed a tool-supported process to create a feature model from tabular specifications [34]. They assumed input was already in tabular form before normalizing these requirements into a more structured format for interpretation and synthesizing the feature model based on high-level defined directives. In contrast, we gather the variability requirements into a central Requirements Traceability Worksheet (RTW) for metadata augmentation to facilitate traceability between the requirements and the feature model. We also leverage a popular open-source feature modeling tool for constraint analysis, which flight software developers are more likely to accept than an unfamiliar tool.

More broadly, Gazzillo and Cohen recently urged that configurability be promoted to a first class element, noting that “configurable software makes up most of the software in use today.” They cite a need for common ground to “bring together researchers and practitioners who are typically siloed” [35]. We hope that the open-source NASA flight software framework may provide one such opportunity for joint innovation.

D. Future Work

A long-range goal is to include VarCORE in the cFS tool suite. Toward this goal we plan to automate additional portions of VarCORE, perhaps using existing NLP techniques [36]–[38], and to enable variability code auto-analysis [39], [40] to evaluate the consistency between requirements specification and implementation.

VII. CONCLUSION

This paper has described our experience with the variability-related requirements and requirement constraints for a configurable NASA flight software framework. A challenge for new projects using this framework is that the specification of variability requirements and constraints are dispersed across multiple documents. This can contribute to requirements inconsistencies, omissions, and conflicts, as well as to inadvertent violations of constraints on combinations of options. We described VarCORE, a new structured technique to create a variability model for the configurable flight software using feature-modeling and open-source tools. We showed in our application of it to a critical piece of the flight software how VarCORE centralizes, specifies, and analyzes the variability requirements and constraints that are currently dispersed. We reported its effectiveness in finding some missing and inconsistent variability requirements and constraints. Finally, we discussed the broader lessons learned from our experience that may be useful to requirements teams on other large configurable industrial projects.

ACKNOWLEDGMENT

We thank the reviewers for their helpful feedback.

REFERENCES

[1] D. Batory, *Automated Software Design Volume 1*, 2nd ed., 2021, pp. 210–220. [Online]. Available: <https://www.cs.utexas.edu/users/dsb/Vol1/index.html>

[2] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[3] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, “Where do configuration constraints stem from? an extraction approach and an empirical study,” *IEEE Trans. Software Eng.*, vol. 41, no. 8, pp. 820–841, 2015.

[4] M. Cashman, M. B. Cohen, P. Ranjan, and R. W. Cottingham, “Navigating the maze: the impact of configurability in bioinformatics software,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*. ACM, 2018, pp. 757–767.

[5] S. Apel, J. M. Atlee, L. Baresi, and P. Zave, “Feature interactions: the next generation (Dagstuhl seminar 14281),” in *Dagstuhl Reports*, vol. 4, no. 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[6] L. R. Soares, P. Schobbens, I. do Carmo Machado, and E. S. de Almeida, “Feature interaction in software product line engineering: A systematic mapping study,” *Inf. Softw. Technol.*, vol. 98, pp. 44–58, 2018.

[7] T. Berger and S. Nadi, “Variability models in large-scale systems: A study and a reverse-engineering technique,” in *Software Engineering & Management 2015. Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17.*, ser. LNI, vol. P-239. GI, 2015, pp. 80–81.

[8] D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, and M. Bartholomew, “Verifying architectural design rules of the flight software product line,” in *13th International Software Product Line Conference*, 2009.

[9] D. Ganesan, M. Lindvall, D. McComas, M. Bartholomew, S. Slegel, B. Medina, R. Krikhaar, C. Verhoef, and L. P. Montgomery, “An analysis of unit tests of a flight software product line,” *Science of Computer Programming*, vol. 78, no. 12, pp. 2360–2380, 2013.

[10] D. J. F. Miranda, “A ‘new space’ approach on spacecraft flight software development using NASA cFS framework,” Master’s thesis, 2019.

[11] “core Flight System: A paradigm shift in flight software development,” <https://cfs.gsfc.nasa.gov/>.

[12] D. McComas, J. Wilmot, and A. Cudmore, “The core flight system (cFS) community: Providing low cost solutions for small spacecraft,” in *Annual AIAA/USU Conference on Small Satellites*, no. GSFC-E-DAA-TN33786, 2016.

[13] “Goddard’s Core Flight Software Chosen for NASA’s Lunar Gateway,” <https://www.nasa.gov/feature/goddard/2021/core-flight-software-chosen-for-lunar-gateway>, [Online; accessed 27-March-2023].

[14] A. van Lamsweerde, *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[15] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, “What is a feature?: a qualitative study of features in industrial software product lines,” in *Proceedings of the 19th International Conference on Software Product Line, SPLC*. ACM, 2015, pp. 16–25.

[16] S. Apel, D. S. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

[17] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger, “Presence-condition simplification in highly configurable systems,” in *37th IEEE/ACM International Conference on Software Engineering, ICSE, Volume 1*. IEEE Computer Society, 2015, pp. 178–188.

[18] T. Ngo, “NASA class A certification of core flight software (cFS),” in *Flight Software Workshop*, 2021. [Online]. Available: <https://www.youtube.com/watch?v=H7WuUxJeAsclist=PLKT7jlj6zb0ALxjewsZ3amBTHPXH7qCindex=36pp=iAQB>

[19] D. McComas, “OpenSatKit-a flight software system educational platform,” 2021.

[20] E. J. Timmons, “Core flight system (cFS) training,” 2020.

[21] “core Flight Executive Software Requirements Specification,” <https://github.com/nasa/cFE>.

[22] IEEE, “IEEE recommended practice for software requirements specifications. IEEE standard, 830-1998.” Institute of Electrical and Electronics Engineers New York, 1998.

[23] T. C. Lethbridge, J. Singer, and A. Forward, “How software engineers use documentation: The state of the practice,” *IEEE software*, vol. 20, no. 6, pp. 35–39, 2003.

[24] “Goldilocks principle,” https://en.wikipedia.org/wiki/Goldilocks_principle, [Online; accessed 26-March-2023].

[25] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund, “Abstract features in feature modeling,” in *15th International Software Product Line Conference*. IEEE, 2011, pp. 191–200.

[26] J. Cleland-Huang, M. P. E. Heimdal, J. H. Hayes, R. R. Lutz, and P. Maeder, “Trace queries for safety requirements in high assurance systems,” in *Requirements Engineering: Foundation for Software Quality - 18th International Working Conference, REFSQ Proceedings*, ser. Lecture Notes in Computer Science, vol. 7195. Springer, 2012, pp. 179–193.

[27] R. L. Epstein, *The Semantic Foundations of Logic Volume 1: Propositional Logics*. Springer Science & Business Media, 2013, vol. 35.

[28] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study,” Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, Tech. Rep., 1990.

[29] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering software variability with FeatureIDE*. Springer, 2017.

[30] K. Schmid, R. Rabiser, M. Becker, G. Botterweck, M. Galster, I. Groher, and D. Weijns, “Bridging the gap: voices from industry and research on industrial relevance of SPLC,” in *SPLC ’21: 25th ACM International Systems and Software Product Line Conference, Volume A*. ACM, 2021, pp. 184–189.

[31] J. Krüger, G. Çalikli, T. Berger, T. Leich, and G. Saake, “Effects of explicit feature traceability on program comprehension,” in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. ACM, 2019, pp. 338–349.

[32] M. Chechik, I. Stavropoulou, C. Disenfeld, and J. Rubin, “FPH: efficient non-commutativity analysis of feature-based systems,” in *Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, ETAPS*, ser. Lecture Notes in Computer Science, vol. 10802. Springer, 2018, pp. 319–336.

[33] A. Rajbhoj, P. Nistala, V. Kulkarni, S. Soni, and A. Pathan, “DizSpec: Digitalization of requirements specification documents to automate traceability and impact analysis,” in *30th IEEE International Requirements Engineering Conference, RE*. IEEE, 2022, pp. 243–254.

[34] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire, “On extracting feature models from product descriptions,” in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-intensive Systems*, 2012, pp. 45–54.

[35] P. Gazzillo and M. B. Cohen, “Bringing together configuration research: Towards a common ground,” in *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!* ACM, 2022, pp. 259–269.

[36] S. Abualhaija, C. Arora, M. Sabetzadeh, L. C. Briand, and E. Vaz, “A machine learning-based approach for demarcating requirements in textual specifications,” in *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE, 2019, pp. 51–62.

[37] P. V. Nistala, A. Rajbhoj, V. Kulkarni, S. Soni, K. V. Nori, and R. Reddy, “Towards digitalization of requirements: generating context-sensitive user stories from diverse specifications,” *Automated Software Engineering*, vol. 29, no. 1, p. 26, 2022.

[38] J. Zhang, S. Chen, J. Hua, N. Niu, and C. Liu, “Automatic terminology extraction and ranking for feature modeling,” in *IEEE 30th International Requirements Engineering Conference (RE)*. IEEE, 2022, pp. 51–63.

[39] Z. Patterson, Z. Zhang, B. Pappas, S. Wei, and P. Gazzillo, “SugarC: scalable desugaring of real-world preprocessor usage into pure C,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2056–2067.

[40] P. D. Schubert, P. Gazzillo, Z. Patterson, J. Braha, F. Schiebel, B. Hermann, S. Wei, and E. Bodden, “Static data-flow analysis for software product lines in C: Revoking the preprocessor’s special role,” *Automated Software Engineering*, vol. 29, no. 1, p. 35, 2022.