Enhancing the requirements engineering of configurable systems by the ongoing use of variability models

Abstract

Software systems and product lines often use configurable features to specify a portfolio of product variants from a common core. Typically, their requirements also include constraints on which combinations of features are valid. Especially for larger systems and systems where the specifications are scattered among documents, the analysis of a new product's variability-related requirements is challenging. To address this, we introduce a scalable, tool-supported framework that uses a variability model to automate checks for missing and inconsistent features and constraints. Our approach also extends and scales traditional variability requirements engineering by incorporating combinatorial interaction testing techniques to build valid product variants covering all configurations in the variability model and to automatically discover faulty feature settings in failed builds. Results from evaluation on two configurable systems show that our framework is effective both at early detection of missing, incorrect, and inconsistent variability requirements and at later finding faulty feature configurations.

Keywords: Requirement engineering, Variability constraints, Variability requirements, Configurable system, Variability model, Combinatorial interaction testing

1 Introduction

Configurable software systems and product lines use variability requirements to specify a portfolio of product variants beyond a common core. Variability requirements include both configurable features, which specify options, and variability constraints, which specify which combinations of features are valid [1–3]. An ongoing challenge for requirements analysts is how to detect missing and inconsistent variability-related features and constraints. Rarely is the software so small that all combinations of configurable features can be checked.

The work reported here was motivated by, and builds on, earlier experience with a NASA configurable flight-software system used by multiple space missions. For new users wanting to reuse this flight software in their own products (such as small satellites), it can be cumbersome to figure out what options are available and

what combinations of options are not allowed, especially since the variability requirements are spread across several documents. Based on experience with its configurable software requirements, the authors proposed addition of a new software artifact, a variability model, to support easier analysis by future developers of the flight software in a new product [4]. A variability model is a higher-level representation of features, dependencies, and constraints among them [5].

In this paper we report our development of a new and significantly more general framework, VarCORE$^+$, that is both scalable to highly configurable systems [6] and appropriate for a wide range of application domains. VarCORE$^+$ is a tool-supported framework that uses a variability model to automate checks for missing and inconsistent features and constraints. Our approach extends and scales traditional variability requirements engineering by incorporating combinatorial interaction testing techniques to build valid product variants covering all configurations in the variability model and to automatically discover faulty feature settings in failed builds.

The new work described here provides new variability modeling and analysis capabilities that help detect variability-related issues. It continues to be the case that many projects only discover configuration constraints that are inconsistent or missing from the requirements during testing or operations [7]. We seek to discover these variability issues earlier and to provide projects with the traceability information they need to resolve them.

While many techniques exist to formally model and analyze all valid configurations exist [8, 9], many industrial projects do not have, nor want to have, the formal models that these techniques need. We aim to meet such projects where they are. Thus, much of our framework has been newly automated to simplify its use and customization by a variety of project domains, and is made available. A limitation is that VarCORE$^+$ takes as input a manually populated requirements traceability worksheet and a mapping between features in the variability model and configurable variables in the code, an artifact which many projects already produce. Use of VarCORE$^+$ on a new project also may require some one-time, low-overhead customization to match its build system or configuration framework. Additionally, this paper reports results from a new application of our tool-supported framework on a configurable software system that shows its effectiveness at scale.

The new contributions of this paper are fourfold:

- Fully automate creation of a variability model in order to support a large number of variability requirements and constraints
- Enhance scalability by building product variants automatically using combinatorial interaction testing (CIT) techniques
- Analyze build results for product variants to automatically identify faulty feature configurations.
- Evaluate VarCORE$^+$, our tool-supported framework, on configurable systems in two different domains

Results reported here from evaluation on two diverse, configurable software systems show that VarCORE$^+$ found multiple variability-requirements issues. The issue types included missing configurable feature, missing configuration constraint, conflict

between configuration constraints, variability requirement specified but not implemented, variability bug occurring only in a single configuration, and variability bug caused by a 3-way feature interaction.

The rest of the paper is organized as follows. Section 2 describes motivation, an illustrative example, and background. Section 3 presents our approach. Section 4 describes our applications and evaluation of VarCORE$^+$. Section 5 discusses findings, threats to validity, related work, and future work. Section 6 provides concluding remarks.

2 Motivation and Background

In this section we first motivate our work, then demonstrate the usage of VarCORE$^+$ on a simple car product, and lastly provide background information on two real-world applications and a combinatorial interaction testing technique used in our approach and evaluation.

2.1 Motivation

Our work is motivated in large part by experience with an open-source configurable NASA flight software system. The flight software uses many best practices for variability management and is well maintained by experienced developers. However, for developers initially unfamiliar with it and wanting to reuse it, there remain obstacles to navigating the maze of information [7] to get their project's variability requirements right. As with many other large, configurable software systems, information about variability-related requirements and constraints is 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 variability requirements or to inadvertently violate constraints by introducing inconsistent or conflicting requirements.

These challenges for the requirements analysis of configurable software systems are well-known. An interview-based study in 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" [10]. Similarly, a survey of industrial practice in variability modeling found that the evolution and visualization of variability models were the most frequently reported problems [11]. Pointing the way to potential solutions, 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 [12]. More recently, Ruiz et al. reported that practitioners do value traceability despite it still being "mainly performed manually" [13].

Variability-related requirements and constraints are easily overlooked and need special attention, especially in high-dependability configurable software. Inconsistent, invalid, or missing variability requirements and constraints have been contributing causes to multiple spacecraft anomalies, many with safety implications for the system or mission[14–16]. Our aim in the work reported here has been to improve the state of requirements engineering practice for configurable software systems or product lines.

VarCORE$^+$ makes the knowledge about variability requirements and constraints on their valid feature combinations less tacit and more obvious to the developers.

We anticipate that VarCORE$^+$ may be especially useful to developers in finding inconsistencies or omissions of requirements in abnormal scenarios. An example is error-recovery software, which is triggered only when something goes wrong. In safety-critical systems, recovery may be essential to safe operation or mission completion. In prior work on spacecraft software, violation of requirements constraints were found to be more common in error-recovery scenarios than in normal scenarios [17]. Error-recovery scenarios typically involve critical coordination among software processes, including some interactions that do not occur in normal scenarios, so may be less understood. Moreover, because error-recovery scenarios occur less often, these atypical interactions may not have been as thoroughly thought out or verified. VarCORE$^+$ can contribute by helping developers visualize and verify that variability requirements and constraints are implemented in error-recovery and other critical but rarely used software. For spacecraft, this includes launch, planetary orbital insertion, and fault-protection software. Spacecraft and other configurable software systems with long operational lifetimes may experience turnover in personnel and ensuing loss of knowledge about dependencies and constraints among options. For long-lived configurable systems, VarCORE$^+$ may offer another tool toward maintaining safe operations over time.

2.2 Example

Consider the requirements (see Table 1) for a simple car product.

Req ID	Description
1	A car shall include a model
2	A car model shall select one and only one of low end or high end model
3	A high end car shall include camera sensors
4	A car may include autopilot

Table 1: Textual Requirements for a Simple Car

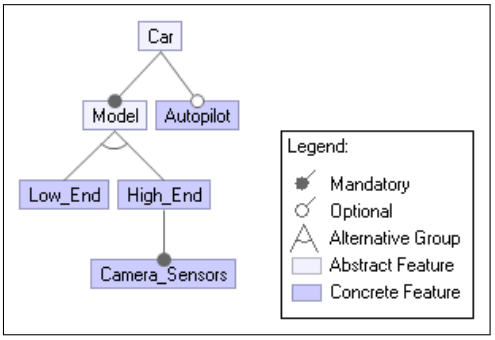


Fig. 1: Variability Model for a Simple Car

Constructing and analyzing a variability model. Variability requirements may be dispersed across multiple requirements documents. We propose to collect variability requirements into a centralized structured worksheet (see Section 3.2 for details). This eases the transformation of the textual variability requirements into an easy-to-read and understand graphical representation. Fig 1 is the graphical variability model automatically generated by VarCORE$^+$ based on the textual requirements in Table 1. VarCORE$^+$'s static analysis flags any invalid (dead or false-optional) elements and

any inconsistencies among the variability requirements. Here, it finds that Fig 1 is a valid variability model.

Iterating and tracing as requirements evolve. Suppose that two updates now are made to the requirements. First, the autopilot is updated to use camera sensors, for example to detect stop lights and signs. This functionality triggers a new constraint in the car's requirements: "autopilot requires camera sensors" (Req ID 5 in Table 2) Second, at the same time the new autopilot will newly be included in all models of cars This changes requirement Req ID 4 (in Table 2) as the autopilot is now a mandatory feature instead of an optional feature.

Req ID	Description
1	A car shall include a model
2	A car model shall select one and only one of low end or high end model
3	A high end car shall include camera sensors
4	A car ~~may~~ shall include autopilot (change)
5	Autopilot requires camera sensors (new)

Table 2: Evolved Car Variability Requirements

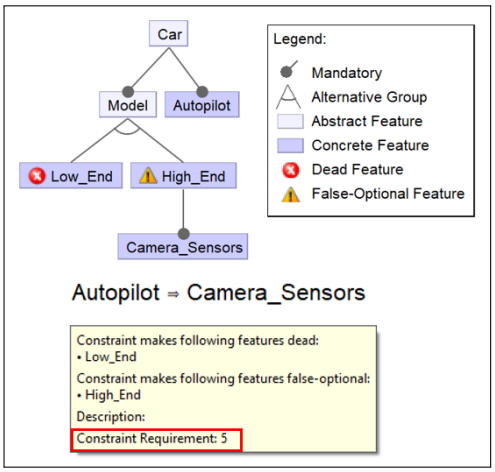


Fig. 2: Variability Model with Evolved Car Requirements

At each requirements update, the developers can use VarCORE$^+$ to automatically regenerate the new variability model and perform static analysis on it to verify its validity and consistency. VarCORE$^+$ now finds that the constraint requirement Req ID 5 makes the feature *Low_End* a dead feature (since it has no camera sensors it cannot be selected). It also finds that the feature *High_End* is now a false-optional feature (since it must be selected to satisfy the requirements). Since VarCORE$^+$ includes traceability between requirements and features, Fig. 2 provides the trace (highlighted in the red box) to the constraint causing the model inconsistency.

To fix it, the developer updates the requirements, shown in Table 3. Thus, Req ID 3: "A high end car shall include camera sensors" is removed and replaced by the new Req ID 6: "A car shall include camera sensors" (see Table 3). Since autopilot and camera sensors are now the mandatory features for all cars, the constraint: "Autopilot requires camera sensors" (Req ID 5 in Table 3) becomes a redundant constraint, and is flagged as such in the output. However, redundant constraints do not break the model, and here it is a good practice to keep this constraint explicit as it reinforces awareness of the autopilot's dependency on the camera sensors. After making the corrections in

requirements, VarCORE$^+$ is used to regenerate and verify the new variability model (see Fig. 3).

Req ID	Description
1	A car shall include a model
2	A car model shall select one and only one of low end or high end model
3	~~A high end car shall include camera sensors~~ (delete)
4	A car shall include autopilot (change)
5	Autopilot requires camera sensors (new)
6	A car shall include camera sensors (new)

Table 3: Revised Variability Requirements for Car

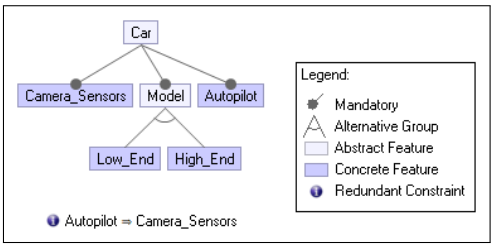


Fig. 3: Variability Model with Consistent Requirements

Generating minimum sets of valid configurations to cover feature combinations. In the simple car example above (see Fig. 1), the variability model generates four valid configurations: {Low_End}, {Low_End, Autopilot}, {High_End, Camera_Sensors}, and {High_End, Camera_Sensors, Autopilot}. These are the only valid configurations out of the possible $2^4 = 16$ configurations that satisfy all four requirements in Table 1.

As the number of configurations grows, identifying all valid configurations from the textual variability requirements and constraints manually is not an easy task, and it is easy to make mistakes. VarCORE$^+$ computes and automatically produces the valid configurations once the variability model is constructed and checked. However, for configurable systems or product lines with a large number of features, VarCORE$^+$ also includes combinatorial interaction testing (CIT) techniques to generate minimum sets of configurations that cover all possible t-way combinations of features instead of finding all valid configurations. For N boolean configurable features, CIT can significantly reduce the configuration space from 2^N to a much smaller space, which is proportional to $2^t \log N$ [18] (where t is the degree of combinatorial interactions, see Section 2.5). By default, VarCORE$^+$ automatically produces minimum sets of configurations that cover all possible 2-way (i.e. pair-wise) combinations of features. (Users can change the t-way to higher degrees of combinatorial interactions if desired.)

For faulty configurations that fail to build, VarCORE$^+$ automatically identifies the faulty feature or combination of features that cause each failed configuration for further investigation by the developer/tester. It also provides them with traces to the requirements associated with each involved feature or combination of features to expedite their resolution.

2.3 Core Flight System (cFS)

The National Aeronautics and Space Administration (NASA) core Flight System (cFS) [19] is an open-source configurable flight software that can support the majority of basic flight requirements for a variety of spacecraft missions. It was created by experienced developers based on past successful spacecraft software.

In 2020, cFS[1] 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 Hopkins University Applied Physics Lab (APL) for all future embedded flight software projects [19]. As part of the Lunar Gateway program, work is also underway to certify cFS as suitable for human-rated vehicles [20]. 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" [21]. Examples include the CubeSat nanosatellites and small spacecraft [22]. Educationally, OpenSatKit (OSK) adopts cFS to provide a free flight software system platform for aerospace and STEM education [23]. To the best of our knowledge, it has not been used by the requirements engineering community, although it has been studied in both architecture and testing papers [24–26].

TIME is the most configurable service module in the core Flight Executive (cFE), which is a framework component of cFS. TIME consists of 14 configurable features, as shown in Table 4, to support various time configurations such as time format (TAI or UTC), time mode (Server or Client), and time-tone order (data following tone or data preceding tone).

To provide some context, Fig. 4 shows a cFS-based, multi-processor flight system with its connectivity and communication channels among flight processors and the ground system. TIME (which resides inside the cFE Fig. 4) manages and distributes the spacecraft time among various mission applications for synchronization in this complex cyber-physical system.

Table 4: TIME Configuration Variability and its Functionality

Variability	**Functionality**
Time_Server	Time operation in Server mode
Time_Client	Time operation in Client mode
Big_Endian_Byte_Order	Force tone message in big endian order
Virtual_MET	Configure as virtual MET if there is no local hardware MET. *virtual_met = false* indicates local hardware MET is enabled.
External_Time_Source	Source of time data is external
MET	Type of external time data source is MET
GPS	Type of external time data source is GPS
Spacecraft_Time	Type of external time data source is spacecraft time
Active_Tone_Signal	Select the active tone signal
TAI_Format	Default time format is International Atomic Time
UTC_Format	Default time format is Coordinated Universal Time
Fake_Tone	Enable fake tone signal generation in the absence of real hardware signal
Data_Following_Tone	Tone signal arrives before "time at the tone" data
Data_Preceding_Tone	Tone signal arrives after "time at the tone" data

[1] https://github.com/nasa/cFS

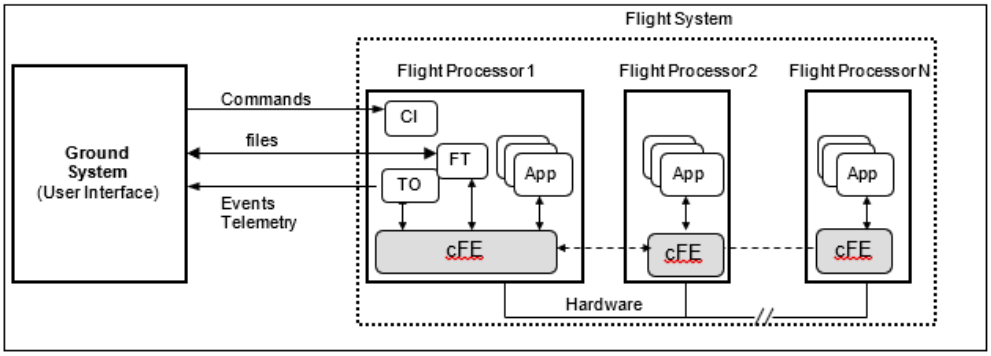


Fig. 4: cFS-based Multi-processor Flight System [27]

In general, TIME uses C-preprocessor (CPP) macros (#define) to define all 14 configurable features. To manage the various combinations of feature configuration within a single module, TIME uses annotated C-pre-processor directive conditionals (such as *#if, #ifdef, #elif, #ifndef*) to control and compile the code based on the setting defined by each configurable feature. Therefore, the configuration of TIME variabilities is defined at build-time and dictates the TIME system behavior at run-time.

2.4 axTLS Embedded Secure Sockets Layer (SSL) Project

The axTLS[2] embedded SSL project is described as a "highly configurable client/server Transport Layer Security (TLS) v1.2 library designed for platforms with small memory requirements" [28]. It is a real-world application which has been used on multiple variability research projects, including SugerC [29] and Kmax [30].

The axTLS uses the KConfig system to construct and select its configurable modules and features at build time. KConfig was designed by developers of large industrial systems [31] and is also used by Linux to configure its kernel. The set of axTLS's configuration options is organized in a tree structure, and its model structure is quite similar to a feature model (described in Section 3.3).

There are 62 boolean configurable features implemented in the axTLS-2.1.5 release. Generation of all valid configurations ($\mathcal{O}(2^N)$ where N is the number of configurable features) for axTLS is not practical. It requires an effective technique to produce a number of configurations feasible for use in analysis and testing.

2.5 Combinatorial Interaction Testing (CIT)

According to empirical studies by the National Institute of Standards and Technology (NIST) and others, all software failures could be triggered by the interaction of one to six variables (i.e., configurable features) [18]. The studies included applications in a variety of domains such as embedded medical devices, web browser, HTTP server,

[2] https://axtls.sourceforge.net/

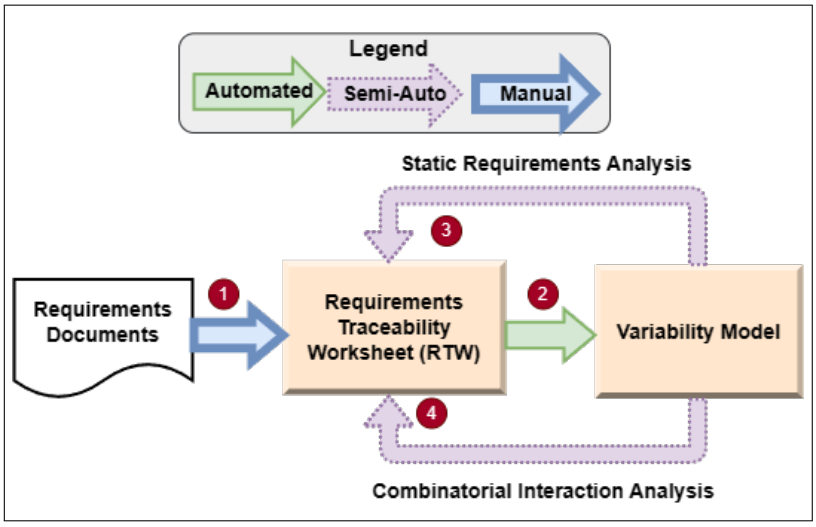


Fig. 5: VarCORE$^+$ Overview.

network security, and aerospace. For instance, in the study of various NASA applications, "67% of the failures were triggered by only a single parameter value, 93% by 2-way combinations, 98% by 3-way combinations and reaching 100% with 4-way interactions" [18]. This finding indicated that testing up to 4-way combinations of configurable features can be highly effective for detecting software failures.

CIT is an innovative testing technique that generates sets of configurations covering t-way combinations of configurable features. CIT is proven effective [18, 32–34] in increasing the effectiveness of software testing for many applications with lower testing effort. It is often adopted in software testing of highly configurable systems and software product lines.

For N boolean configurable features, CIT can significantly reduce the configuration space from 2^N to a much smaller space, proportional to 2^t log N [18] (where t is the degree of combinatorial interactions). Therefore, even with a large number of configurable features, CIT may still produce a feasible number of configurations for analysis and testing. For instance, in a system with 10 boolean configurable features, instead of generating all 1024 (2^{10}) configurations, CIT can reduce the configurations to just 13 that cover all possible 3-way ($t = 3$) combinations of features [18].

3 Approach

In this section we describe our VarCORE$^+$ framework and the new software variability model it produces. The framework uses combinatorial interaction testing (CIT) techniques to generate a reduced set of product variants for verifying the compliance of the implementation to the variability requirements.

3.1 Overview

Fig. 5 gives an overview of VarCORE$^+$ and the two main artifacts[3] (Requirements Traceability Worksheet (RTW) and Variability Model) it provides. First, VarCORE$^+$ collects variability requirements into a centralized structural worksheet. This is done manually by the requirements content expert. Second, VarCORE$^+$ constructs the variability model automatically using information from the worksheet. Then, VarCORE$^+$ performs automatic static analysis to check the validity and consistency of the variability requirements. The user is required to import the generated variability model to a tool and analyze the result generated automatically by the tool. Finally, VarCORE$^+$ conducts semi-automated combinatorial interaction analysis to verify that the generated product variants are consistent with the variability requirements. This section describes the details of our approach.

3.2 Variability Requirements Collection

The primary input to VarCORE$^+$ is the software requirements documents. For large-scale software projects there are typically more than one requirements document, starting from system requirements and decomposing them into subsystem, module and atomic 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 requirements documents. In our approach, we propose a lightweight process to collect the scattered variability requirements from diverse requirements documents into a centralized Requirements Traceability Worksheet (RTW), illustrated in Table 5.

Table 5: Requirements Traceability Worksheet (RTW)

Requirements Traceability Worksheet (RTW)						
Requirements ID	**Requirements Specification**	**Valid**	**Rule**	**Parent Feature**	**Children Features**	**Source of Requirements**

First, the variability requirements are extracted from the dispersed requirements documents. From each extracted requirement or constraint, we identify the relevant variant features (i.e., units of functionality) and their hierarchical relationships such as parent-child or constraint relationship. Then we standardize the structure of each requirement or constraint according to the format specified in Table 6, based on the features' relationship. To facilitate traceability, each requirement or constraint is assigned a unique requirement ID, and the source of each requirement is logged. Finally, the requirements ID, the structured requirements specification, the features' relationship, the parent features, the children features and the source of the requirements are entered into the RTW, one requirement or constraint per entry.

[3] https://github.com/chinkhor/VarCORE2/tree/main/artifacts

Table 6: Requirements Specification Template

Features Hierarchical Relationship	**Requirements Specification Format**
Parent and a mandatory child	'Parent feature' **shall** <action verb> 'child feature'
Parent and an optional child	'Parent feature' **may** <action verb> 'child feature'
Parent and one and only one of children	'Parent feature' **shall** <action verb> **one and only one of** 'child feature 1', 'child feature 2', ... **or** 'child feature N'
Parent and one or more children	'Parent feature' **shall** <action verb> **at least one of** 'child feature 1', 'child feature 2', ... **or** 'child feature N'
Constraint between two unrelated features	'Feature A' **requires** 'feature B' 'Feature A' **excludes** 'feature B'

The feature relationship is decoded and mapped to the assigned rule as shown in Fig. 6 and described below, with the relevant rule entered into the RTW's Rule field. For a cross-tree constraint (indicated by rule R7) in which the constraint is established between two unrelated features, the first feature is entered as the parent feature and the second feature is entered as child feature. Such constraints "cross-cut" hierarchy dependencies in the variability model [11]. For the Validity field of the RTW, all variability requirements are initially set to valid (represented as a "1"). If subsequent requirements analysis reveals that a RTW entry is invalid, that entry can be invalidated by setting the value in the Valid field to zero. VarCORE$^+$ discards all invalidated entries during the construction of variability model.

VarCORE$^+$ provides a utility to export the valid RTW entries into a Comma Separated Values (CSV) format file. This eases the integration of the variability requirements in the RTW into widely used requirements management tools such as IBM Rational DOORS [35] or JIRA Software [36].

3.3 Variability Modeling

Once the RTW has been created, VarCORE$^+$ will automatically generate a variability model (i.e., represented by a feature model). Feature models are a standard type of variability model that are widely used in software product lines (SPLs) [11, 37, 38]. A feature model is a feature diagram that represents the features of the software product line in a graphical view, together with the constraints on those features. Features are units of functionality, i.e., functional requirements [37, 39, 40]. Feature models were first introduced by Kang et al. [41] in their Feature Oriented Domain Analysis (FODA) study in 1990. Many researchers have studied the relationship between FODA notation and propositional logic. Fig. 6 summarizes the mapping between the propositional logic and FODA notation which was provided by Batory [1]. Each mapping is identified here using a rule ID.

To construct a variability model, VarCORE$^+$ extracts the features from each RTW entry and uses them as feature nodes to build a feature tree. The connectivity of feature nodes is determined by their hierarchical parent-child relationship as specified in the RTW (indicated by the Rule field). VarCORE$^+$ then performs a breadth-first search to examine the feature connectivity in the feature tree. Any unconnected feature

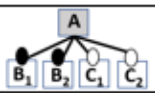
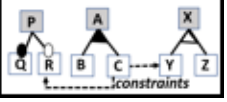
Rules	Propositional Formula	Semantics (Hierarchical Relationship)	FODA Notation
R1	root	Root (mandatory)	
R2	$A \Leftrightarrow B$	Select Mandatory Child	
R3	$C \rightarrow A$	Select Optional Child	
R4	$A \Leftrightarrow (B \wedge \neg C) \vee (\neg B \wedge C)$	Select Only 1 Child	
R5	$A \Leftrightarrow (B \vee C)$	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	$C \rightarrow Y$ (require) $C \rightarrow \neg R$ (exclude)	Cross Tree Constraints	

Fig. 6: Rule Definition for Mapping Propositional Formulas to FODA

(i.e., a feature that does not appear in the feature tree) is reported for the developers' attention. It may indicate a missing requirement for specifying the relationship of the unconnected feature with the rest of the features in the requirements. Alternatively, it may indicate an inconsistent use of that feature's name in the requirements.

Using the traceability information inherent in VarCORE$^+$'s design, the requirement that specifies the unconnected feature can be identified using its requirement ID, and the source of the requirement can be retrieved from the respective RTW entry for further investigation by the developer. Requirements and constraints for the unconnected features are invalidated and excluded by VarCORE$^+$ from the variability model construction. The developer can remedy the model by adding the missing requirement(s) or by fixing an inconsistently named feature in the RTW and regenerating the variability model.

After assuring that all features are fully connected in the feature tree, VarCORE$^+$ constructs the variability model (also called a feature model) using the syntax specified by FeatureIDE [42], a widely used open-source feature modeling tool. First, VarCORE$^+$ traverses the fully connected feature tree using depth-first search. At every traversed feature, its notation will be determined based on the relationship with its child or children (as indicated by the rule shown in Fig. 6). All variability-related constraints (indicated by rule R7 in the RTW entry) are converted to propositional logic formulas (see Fig. 6) to represent variability model constraints. This intermediate product from VarCORE$^+$ is in the form of Extensible Markup Language (XML). The XML format is fully compatible with the FeatureIDE tool, enabling it to output a visual display of the feature tree and the constraints.

3.4 Static Analysis of Variability Requirements

The variability model constructed by VarCORE$^+$ can be imported into the FeatureIDE tool for feature-diagram rendering and variability model analysis. Fig. 7 shows a sample feature diagram consisting of a hierarchical feature tree and a list of constraints associated with those features. The advantage of the feature diagram is that it is easier to read and understand the relationship among features by viewing the feature diagram than by reading requirements in the textual form.

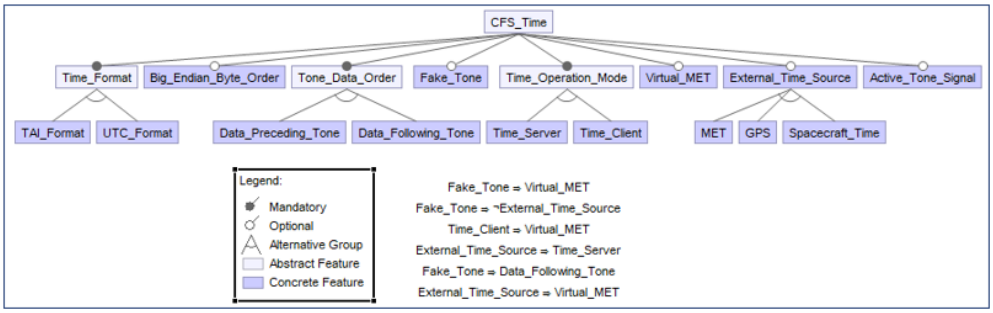


Fig. 7: Variability Model for cFS Time Module

Checking consistency among variability requirements is often a challenging effort. We use FeatureIDE's built-in capabilities to analyze the model's validity, including checks for void-model detection (i.e., a variability 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., an optional feature but present in all products), constraint redundancy, and conflict of constraints. During the variability model analysis, FeatureIDE flags any occurrence of these errors in the feature diagram and includes them in its statistical report.

VarCORE$^+$'s generated variability model has traceable requirement IDs embedded in the description of each feature and constraint (see highlighted red boxes of Fig. 8). This is useful for analyzing and troubleshooting any model inconsistencies and violations by narrowing the investigation of the requirements causing the problem. For example, once the erroneous feature or constraint is reported by FeatureIDE, the analyst can use the requirement IDs associated with each feature or constraint to identify the impacted variability requirements or constraints using RTW. The analyst also can more quickly identify the source of any problematic requirements or constraints (e.g., the requirements document where they originate) by means of their RTW entries.

The requirements analysis may result in changes needing to be made to some requirements. VarCORE$^+$ provides strong support for requirements evolution. Requirements can be added, modified, deleted or invalidated in the RTW at any time. Furthermore, variability-model creation from the RTW using VarCORE$^+$ is fully automated. This means that the RTW can be scaled up or down incrementally as needed. Additionally, VarCORE$^+$ can be used to create experimental variability models for comparison of alternatives, change-impact analysis, or troubleshooting.

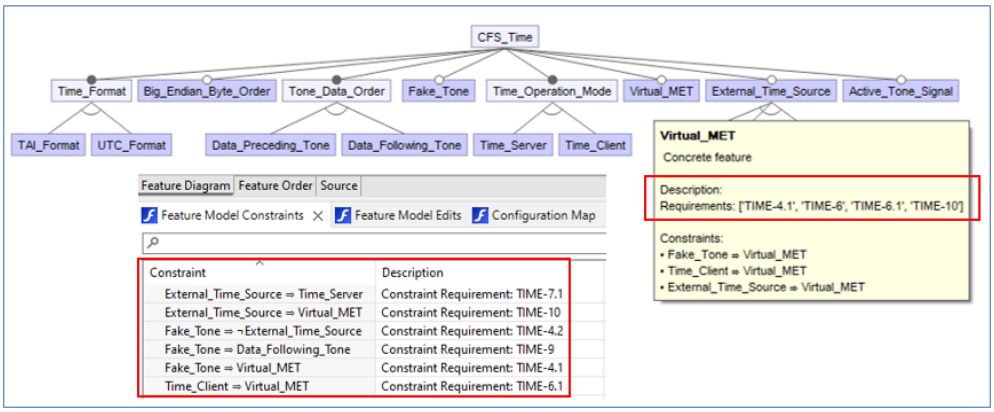


Fig. 8: Variability Model with Traceable Requirements ID

3.5 Combinatorial Interaction Analysis of Variability Requirements

The static analysis described above checks the validity and consistency among the variability requirements. For instance, VarCORE$^+$'s static requirements analysis can find unconnected features indicating missing requirements, and identify the conflicted requirements and/or constraints via variability model analysis. However, the requirements analysis does not detect configuration implementation bugs (i.e., variability requirements not implemented) or variability bugs (i.e., variability requirements not implemented correctly) in the code. Examples include certain feature options that are not implemented, coding errors that change the variability behaviors, and implementation that does not satisfy the variability constraints.

To detect inconsistencies between the specification of the variability requirements in the variability model and the source code, VarCORE$^+$ adds support for combinatorial interaction analysis. It achieves this by using the product configurations generated from the variability model as inputs to the product's build system to create product variants for variability analysis.

Each product configuration generated from the variability model represents a valid configuration that satisfies all variability requirements specified in the RTW. Building these product variants, which can subsequently be handed over to the test team, helps assure consistency between the variability requirements/constraints and the code that should satisfy them. Inconsistencies indicate potential flaws in the requirements or in their implementation.

Fig. 9 expands Fig. 5's combinatorial interaction analysis to illustrate its semi-automated flow in more detail. VarCORE$^+$ uses combinatorial interaction testing (CIT) techniques [18, 32, 33] to generate product variants for verifying the compliance of the implementation to the variability requirements. By default, VarCORE$^+$ uses the state-of-the-art CIT tool: Advanced Combinatorial Testing System (ACTS) [34, 43], shown in blue in Fig. 9, to generate pair-wise (2-way) combinatorial interaction configurations. ACTS is a free tool provided by NIST for constructing t-way

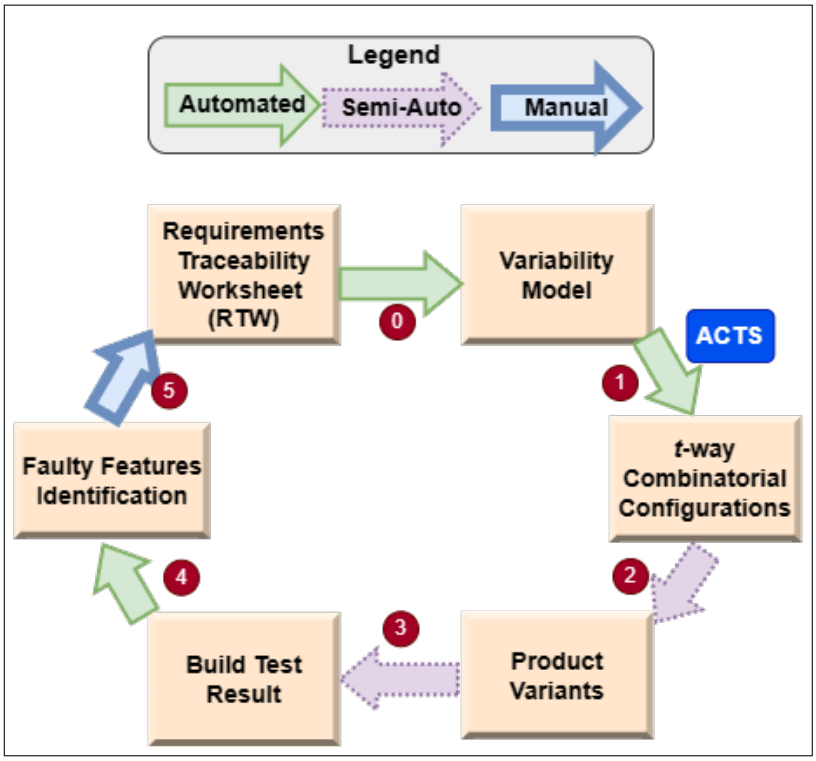


Fig. 9: Overview of VarCORE$^+$'s Combinatorial Interaction Analysis

combinatorial test configurations (where t is the degree of combinatorial interaction). Although VarCORE$^+$ could be readily extended to work with other CIT tools (e.g., CASA [44]), we chose ACTS for its ongoing availability and broad use.

VarCORE$^+$ first (Step 1 in Fig. 9) automatically extracts all features and constraints from the variability model to create an input file for the ACTS tool. The input file consists of all boolean configurable features, and the constraints derived from the variability model (i.e. feature model). VarCORE$^+$ then directs the ACTS tool to generate pair-wise combinatorial interaction configurations based on the provided configurable features and constraints among features using In-Parameter-Order-General-Doubling-Construction (IPOG-D) algorithm [45]. In return, VarCORE$^+$ receives a set of combinatorial configurations in which all possible pairs (2-way) of features are covered by at least one configuration and all specified variability constraints are satisfied. This step is fully automated.

After acquiring the pair-wise combinatorial configurations, VarCORE$^+$ translates the configurations to the respective product's configuration files that are used by the product's build system (Step 2 in Fig. 9) to create the product variants.

Since the feature name used in the variability model may not be identical to the name used in the code implementation, VarCORE$^+$ takes as input a mapping between

the feature variables in the variability model and the configurable variables (e.g., annotated pre-processor(CPP) macros) in the code (see example in Table 7) to correctly perform the translation. The mapping process requires the content expert to perform the matching manually before starting the semi-automated combinatorial interaction analysis. The list of CPP conditionals for the source code (limited to C only) are extracted using the Linux *ifnames*[4] tool to ease the matching with the feature variables in the variability model.

When the mapping is provided, VarCORE$^+$ automatically updates the variability model. Those feature variables which are unmatched are set as abstract features in the updated variability model (i.e feature model). Basically, abstract features represent features that are not implemented in the code and thus will not be used to generate product configuration files. However, they are useful for grouping features to enhance the readability of the variability model. Abstract features make it easier to understand the hierarchical relationship among features (see Fig. 7).

Every project has its own unique build system and configuration framework. A new project will thus manually create the mapping, translate the t-way combinatorial configurations to configuration files, and automate the building of its product variants once (Steps 2 and 3 in Fig. 9). After that, the build system will automatically build the product variants based on the given configuration files. VarCORE$^+$ has followed these steps to automate the product variants' generation for both the cFE TIME service module (see Section 2.3) and for the axTLS embedded SSL project (see Section 2.4).

Table 7: Feature Mapping between Variability Model and Code for cFE TIME

Features in Variability Model	**C Pre-processor Directive Macros in CFE Time Service Module (Code)**
Time_Server	CFE_PLATFORM_TIME_CFG_SERVER
Time_lient	CFE_PLATFORM_TIME_CFG_CLIENT
Big_Endian_Byte_Order	CFE_PLATFORM_TIME_CFG_BIGENDIAN
Virtual_MET	CFE_PLATFORM_TIME_CFG_VIRTUAL
External_Time_Source	CFE_PLATFORM_TIME_CFG_SOURCE
MET	CFE_PLATFORM_TIME_CFG_SRC_MET
GPS	CFE_PLATFORM_TIME_CFG_SRC_GPS
Spacecraft_Time	CFE_PLATFORM_TIME_CFG_SRC_TIME
Active_Tone_Signal	CFE_PLATFORM_TIME_CFG_SIGNAL
TAI_Format	CFE_MISSION_TIME_CFG_DEFAULT_TAI
UTC_Format	CFE_MISSION_TIME_CFG_DEFAULT_UTC
Fake_Tone	CFE_MISSION_TIME_CFG_FAKE_TONE
Data_Following_Tone	CFE_MISSION_TIME_AT_TONE_WAS
Data_Preceding_Tone	CFE_MISSION_TIME_AT_TONE_WILL_BE

VarCORE$^+$ then performs build tests for (only) the product variants. The build test results for the product variants are collected for analysis (Step 3 in Fig. 9). If the product variants are built successfully, they can be used as test cases for further product variability testing (e.g., unit testing, integration testing, and system testing). If

[4] https://code.tools/man/1/ifnames2.64/

a build fails, VarCORE$^+$ performs automated configuration fault analysis to find the faulty-feature setting(s) that caused a build failure (Step 4 in Fig. 9). It achieves this by comparing the feature setting(s) of passed and failed product variants' configurations. Each feature setting, pair-wise feature combination, and 3-way feature combination in the failed configurations is determined and logged. The feature setting(s) or combination of feature setting(s) that do not appear in any of the passed configurations are thus identified as potential causes of the build failure. VarCORE$^+$ first checks for single-feature faults (settings), followed by 2-way and 3-way feature-interaction faults.

The automated configuration fault-detection is an approximation method. It uses passed product variants configurations as references for ruling out the good feature setting(s) or combinations of feature setting(s). It depends on the availability of good references. Thus, it may not be able to pinpoint the cause of faulty feature setting(s) if there are insufficient good references. However, it will still reduce the number of potential faulty feature setting(s) to a small list of candidates for further investigation by analysts. This reduces the effort of identifying faulty feature setting(s) especially in a system with very many configurable features.

Since a failed build may reflect a flaw in the requirements or constraints specification, the analyst also can trace backward from the features identified as potential causes of the build failure to help determine a remedy. The automatic configuration fault analysis provides the requirement IDs that are associated with each identified faulty feature setting. Using the requirements traceability captured in the RTW, the developers can trace from the failed build test back to the impacted requirements and/or constraints and check them for flaws.

If the flaw is caused by erroneous requirements or constraints, the developer can manually update the RTW to correct the requirements or constraints (Step 5 in Fig. 9). Then they can use VarCORE$^+$ iteratively to re-generate the variability model automatically (Step 0 in Fig. 9) and re-start the semi-automated combinatorial interaction analysis after each correction (either to the RTW or to the code) until consistency between the requirements/constraints and the implementation is achieved.

4 Applications and Evaluation

In this section we describe our application of VarCORE$^+$ on a portion of the cFS flight software and a real-world highly configurable software (axTLS). We also present our findings regarding the following four research questions:

- RQ1: Did VarCORE$^+$ find missing variability requirement(s)?
- RQ2: Did VarCORE$^+$ find inconsistencies among the specified variability requirements?
- RQ3: Did VarCORE$^+$ find variability requirement(s) not implemented in the code?
- RQ4: Did VarCORE$^+$ scale to support configurable software in both domains?

4.1 System Configuration

The system (hardware and software) that was the setup for the evaluation consists of:

- Intel CORE vPRO i7 processor (with 16GB RAM memory)

- Linux Ubuntu 22.04.2 LTS (with python3.10.12 and javac-11.0.20.1)
- ACTS-3.2[5]
- cFS Caelum release[6]
- axTLS-2.1.5 release[7]
- VarCORE$^+$[8]

4.2 Application and Evaluation on cFE TIME

We chose to evaluate our VarCORE$^+$ technique first on the smaller configurable NASA core Flight Executive (cFE) TIME software. Initially we selected the cFE Software Requirement Specification [27] and the cFE Funtional Requirements specification [46] as the sources from which to collect the variability requirements for cFE TIME. However, we found that only six variability requirements for cFE TIME are described in these two requirements documents. Moreover, they do not document the configuration options available or the variability constraints that the developers of a new product need to know. We then discovered that, instead, the cFE User's Guide (which is the 1,056 page reference for application, tool and test development) describes the TIME variability requirements in detail. We thus used it as the primary input document for our VarCORE$^+$ application and evaluation. It specifies requirements for the configuration options and the variability constraints. However, its length renders it difficult for projects to navigate and use.

Table 8 shows an excerpt of the first generated artifact produced with VarCORE$^+$, the RTW (Requirements Traceability Worksheet) described in Section 3.2. Following the process described there, 21 variability requirements were identified for cFE TIME configurations and extracted from the cFE User's Guide. Then, 18 configurable features (see Figure 7) were derived from the 21 variability requirements.

4.2.1 Static Analysis of Variability Requirements

With cFE TIME's variability requirements now collected into the structured RTW, we used VarCORE$^+$ to construct the variability model automatically. During the construction process, VarCORE$^+$ created the feature tree for connectivity analysis and discovered that feature *Single_Processor* did not appear in the feature tree. Feature *Single_Processor* was used by variability requirement TIME-5.2 (see Table 8) to set constraint with feature *Time_Server*, but it was not defined by any variability requirement. There is thus a missing variability requirement in the RTW for feature *Single_Processor* definition.

Variability requirement TIME-5.2 originated from this constraint statement in the cFE User's Guide[9]: "If the target system has only one processor running the cFE, then TIME must be configured as a server". This is a legitimate constraint. However, there

[5] https://drive.google.com/file/d/1TERK99sNwSrKaUbHfvjojogfdjxqP7cp/view?usp=sharing
[6] https://github.com/nasa/cFS
[7] https://sourceforge.net/projects/axtls/files/
[8] https://github.com/chinkhor/VarCORE2
[9] https://github.com/nasa/cFS/blob/gh-pages/cfe-usersguide.pdf

Table 8: Excerpts from the cFE TIME Requirements Traceability Worksheet (RTW)[47]. Variability requirements that are unsatisfiable or conflicted with others are highlighted in red

Requirements Traceability Worksheet (RTW)						
Req ID	**Requirements Specification**	**Valid**	**Rule**	**Parent Feature**	**Children Features**	**Source of Requirements**
TIME-0	The core Flight System Time (cFS Time) is the time management service module	1	R1	Root	CFS_Time	cfe-userguide, section: 1.22.4 Time Configuration
TIME-5.1	The time operation mode shall use one and only one of the time server or the time client	1	R4	Time_ Operation_ Mode	Time_Server, Time_Client	cfe-userguide, section: 1.22.4.5 Specifying Time Server/Client
TIME-5.2	The single processor requires the timer server	0	R7	Single_ Processor	Time_Server	cfe-userguide, section: 1.22.4.5 Specifying Time Server/Client
TIME-6	The cFS Time may use the virtual MET	1	R3	CFS_Time	Virtual_MET	cfe-userguide, section: 1.22.4.7 Virtual MET
TIME-6.1	The time client requires the virtual MET	1	R7	Time_Client	Virtual_MET	cfe-userguide, section: 1.22.4.7 Virtual MET
TIME-6.2	The time server requires the virtual MET	0	R7	Time_Server	Virtual_MET	cfe-userguide, section: 1.22.4.7 Virtual MET
TIME-7	The cFS Time may use the external time source	1	R3	CFS_Time	External_ Time_Source	cfe-userguide, section: 1.22.4.8 Specifying Time Source
TIME-7.1	The external time source requires the time server	1	R7	External_ Time_Source	Time_Server	cfe-userguide, section: 1.22.4.8 Specifying Time Source
TIME-8	The cFS Time may specify the active tone signal	1	R3	CFS_Time	Active_Tone_ Signal	cfe-userguide, section: 1.22.4.9 Specifying Time Signal
TIME-10	The external time source requires the virtual MET	1	R7	External_ Time_Source	Virtual_MET	cfe-userguide, section: 11.98.2.13 CFE_PLATFORM_ TIME_CFG_VIRTUAL

is no variability requirement for specifying single or multiple processors in the requirement documents. In the code review process, we also did not find any implementation related to single or multiple processors variability in cFE TIME.

In this case, VarCORE$^+$ reported the feature *Single_Processor* as an undefined feature for attention by the developer, then invalidated its associated requirement (e.g. TIME-5.2) to exclude it from variability model construction. In general, VarCORE$^+$ automatically invalidates and excludes all unsatisfiable requirement(s), and uses only the features appearing in the fully connected feature tree to build the variability model.

By default, all identified features from variability requirements are initially set as concrete features (indicating features implemented in code). However, in reality not all features are implemented as needed. We used the *ifnames* tool to list all configurable variables (parameters) in the cFE TIME implementation and performed mapping between features and variables. All four unmatched features were changed to abstract features in the variability model, which indicated that they were not implemented in cFE TIME. These unmatched features include *cFS_Time*, *Time_Format*,

Tone_Data_Order and *Time_Operation_Mode* (see Fig. 7). This update to the variability model served the purpose of improving the readability of the variability model for better understanding of the variability of cFE TIME.

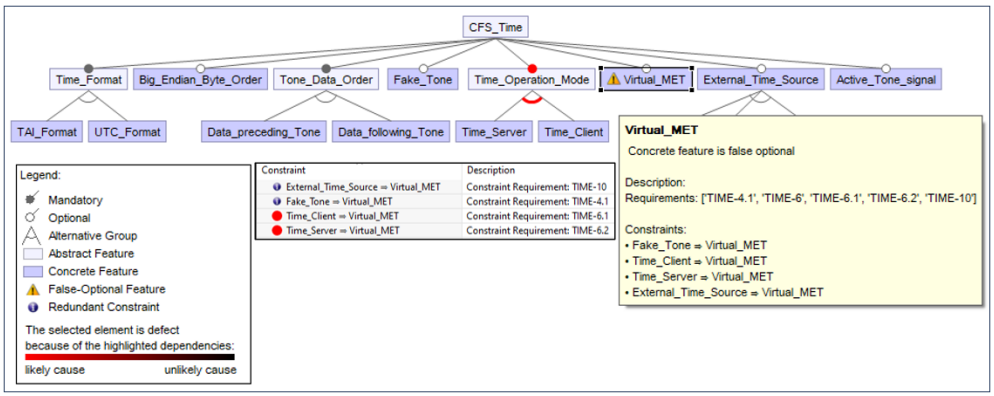


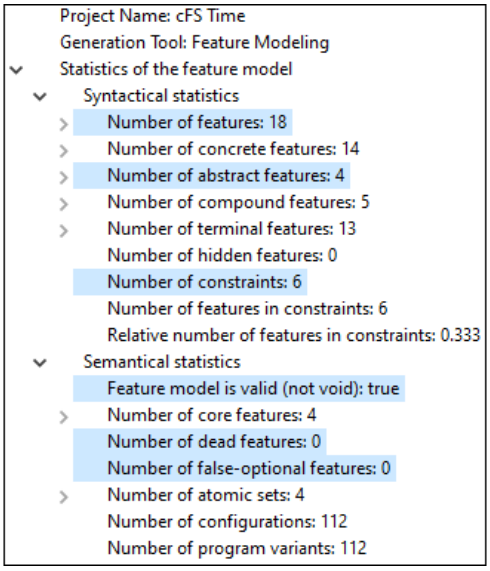
Fig. 10: Initial Constructed Variability Model Showing Detection of False-Optional Feature and Inconsistent Constraints

After producing the variability model, we imported the model into the FeatureIDE tool (described in Sect. 3.4) to render the feature diagram. Fig. 10 shows the *initial* constructed variability model, including the cross-tree constraints. Once the feature diagram was rendered, the FeatureIDE tool automatically checked the model for validity and consistency. As shown in Fig. 10, the FeatureIDE's model checker flagged *Virtual_MET* as a false-optional feature (an optional feature present in all products) and reported that the requirements TIME-6.1 and/or TIME-6.2 were the likely cause of the issue.

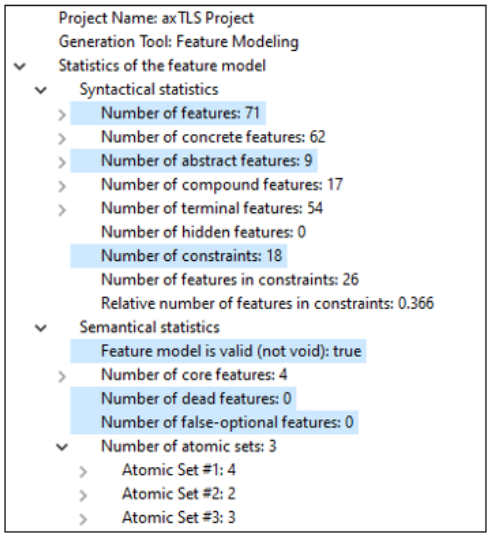
Knowing that *Time_Server* and *Time_Client* were mutually exclusive (see Fig. 10 or requirement TIME-5.1 in Table 8), the co-existence of requirements TIME-6.1 (*Time_Client* $\Rightarrow$ *Virtual_MET*) and TIME-6.2 (*Time_Server* $\Rightarrow$ *Virtual_MET*) would cause the feature *Virtual_MET* to be always selected. This was contradicted by the optional configuration for the feature *Virtual_MET* (see requirement TIME-6 in Table 8 or Fig. 10). Therefore, either requirement TIME-6.1 or TIME-6.2 was invalid.

Using the traceability provided by VarCORE$^+$, we could readily identify the source of constraints TIME-6.1 and TIME-6.2 for further investigation. TIME-6.2 originated in this constraint statement in the cFE User's Guide: "TIME servers must be configured as using a virtual MET". This constraint is contradicted by TIME-6.1 and TIME-6 which require instantiation of TIME to be Server when the MET (Mission Elapsed Time) is local (i.e., *Virtual_MET* = false).

Therefore, the variability constraint specified in TIME-6.2 (highlighted in red in the RTW, see Table 8) was found to be erroneous by the developers. After removal of TIME-6.2, the new generated variability model (see Fig. 7) was consistent and valid without any dead feature or false-optional feature (see Fig. 11a).



(a) cFE TIME



(b) axTLS. Number of configurations and program variants were not computable due to large configuration space

Fig. 11: FeatureIDE Statistics for cFE TIME and axTLS

4.2.2 Combinatorial Interaction Analysis of Variability Requirements

After completion of the variability model analysis, we used VarCORE$^+$ to generate pair-wise (2-way) combinatorial feature interaction configurations, which produced only 13 configurations. All possible pairs (total 449 pairs) of the cFE TIME feature combination were covered at least once by these 13 configurations. VarCORE$^+$ then automatically mapped the concrete features to the respective variables in the code using the mapping table (see Table 7) created earlier. After mapping, all 13 configurations were transformed to code configurations files which were recognized by the cFE TIME build system. Subsequently, VarCORE$^+$ built all 13 product variants, one product variant per code configuration file.

Table 9 shows all 13 generated configurations and their respective build results. 7 of the 13 product variants failed to build successfully. Finally, VarCORE$^+$ analyzed the build results to identify any faulty feature setting(s). Fig. 12 (console output of VarCORE$^+$) shows that two faulty feature settings: $Virtual\_MET = False$ and $Active\_Tone\_Signal = True$ were identified. They were both single-parameter faults which caused all 7 of the cFE TIME product variants' build failures. VarCORE$^+$ did not find any 2-way or 3-way feature interaction that triggered a build failure.

Table 10 compares three build results. The first build result is the current existing cFE TIME build test result using default configuration, in which the cFE TIME was built successfully. The second build result was acquired from our recent study [4], which derived 112 possible valid configurations for cFE TIME and found that 64

Table 9: cFE TIME's 2-way Combinatorial Configurations and Build Test Results

Features	cfg1	cfg2	cfg3	cfg4	cfg5	cfg6	cfg7	cfg8	cfg9	cfg10	cfg11	cfg12	cfg13
CFS_Time	1	1	1	1	1	1	1	1	1	1	1	1	1
Time_Format	1	1	1	1	1	1	1	1	1	1	1	1	1
TAI_Format	0	1	0	1	1	0	0	0	0	0	1	0	1
UTC_Format	1	0	1	0	0	1	1	1	1	1	0	1	0
Big_Endian_Byte_Order	0	1	1	0	0	1	0	1	1	1	0	1	0
Tone_Data_Order	1	1	1	1	1	1	1	1	1	1	1	1	1
Data_Preceding_Tone	0	1	1	0	1	0	0	0	1	0	1	0	1
Data_Following_Tone	1	0	0	1	0	1	1	1	0	1	0	1	0
Fake_Tone	0	0	0	1	0	1	0	1	0	0	0	0	0
Virtual_MET	0	1	0	1	0	1	1	1	1	1	1	1	1
External_Time_Source	0	1	0	0	0	0	1	0	0	1	1	1	1
Time_Operation_Mode	1	1	1	1	1	1	1	1	1	1	1	1	1
Time_Server	1	1	1	0	1	0	1	1	0	1	1	1	1
Time_Client	0	0	0	1	0	1	0	0	1	0	0	0	0
MET	0	1	0	0	0	0	1	0	0	0	0	0	0
GPS	0	0	0	0	0	0	0	0	0	1	1	0	0
Spacecraft_Time	0	0	0	0	0	0	0	0	0	0	0	1	1
Active_Tone_Signal	0	1	1	0	0	1	0	0	0	0	1	1	0
Build Test Results (F:failed, P:passed)	F	F	F	P	F	F	P	P	P	P	F	F	P

```
Test Result:
  passed: ['4', '7', '8', '9', '10', '13']
  failed: ['1', '2', '3', '5', '6', '11', '12']

Failure triggered by single parameter, potential candidates:
  Virtual_MET = False
     Requirements: ['TIME-4.1', 'TIME-6', 'TIME-6.1', 'TIME-10']
  Active_Tone_Signal = True
     Requirements: ['TIME-8']

No failure triggered by 2-way feature interaction

No failure triggered by 3-way feature interaction
```

Fig. 12: cFS TIME Build Test Result Analysis using VarCORE$^+$

out of 112 configurations (product variants) failed to build. The study determined that all 64 build failures were caused by known configuration implementation bugs[10] triggered by two configuration options: *cFE_PLATFORM_TIME_CFG_SIGNAL*=True or *cFE_PLATFORM_TIME_CFG_VIRTUAL*=False. These two configuration options

[10] https://github.com/nasa/cFE/issues/109

Table 10: Comparison of TIME Build Results

Configuration Generation Method	**Passed**	**Failed**	**Total Configurations**
Current cFS TIME Build System (default configuration)	1	0	1
FeatureIDE's Product Generator (all valid configurations) [4]	48	64	112
ACTS's CIT Method (2-way combinatorial interaction configurations)	6	7	13

trigger the cFE TIME to call the unimplemented abstract functions: *OS_SelectTone()*, *OS_SetLocalMET()* and *OS_GetLocalMET()*, and thus failed the compilation.

The third build result in Table 10 summarizes our build results with 13 product variants for cFE TIME (see Table 9). We found that VarCORE$^+$ was able to find the same configuration implementation bugs discovered in [4], but with just 13 configurations. The two identified faulty feature settings, *Virtual_MET*=False and *Active_Tone_Signal*=True, were mapped to the corresponding configuration options: *cFE_PLATFORM_TIME_CFG_SIGNAL*=True and *cFE_PLATFORM_TIME_CFG_VIRTUAL*=False (see Table 7), which triggered a compilation error.

Comparing the build results (see Table 10) indicates that testing with the single default configuration was not sufficient to discover the configuration bugs. Generating all valid configurations for finding configuration bugs is costly, and may not always be practical, especially for systems with a large number of configurable features. VarCORE$^+$ was effective in finding the same configuration bugs with 10X less test efforts for cFE TIME.

The 13 product variants represented valid configurations derived from cFE TIME variability requirements. A build failure would indicate inconsistency between the implementation and the variability requirements. VarCORE$^+$ used its requirements traceability information to retrieve the relevant variability requirement(s) associated with the faulty features (see Fig. 12). This identified that the code did not satisfy the following variability requirements (refer to Table 8 for more information):

- TIME-6 when the feature *Virtual_MET* is False
- TIME-8 when the feature *Active_Tone_Signal* is True

VarCORE$^+$'s semi-automated process of (1) generating 2-way combinatorial configurations from the variability model, (2) transforming and building the respective product variants from the combinatorial configurations, and (3) analyzing product variants' build results to identify faulty feature setting(s) reduces the effort of combinatorial interaction analysis for verifying requirements/constraints. The generated product variants can also be reused for subsequent software verification activities such as unit testing, integration testing and system testing.

4.3 Application and Evaluation on axTLS

VarCORE$^+$ was initially customized to support NASA's cFS; however, its generic design also makes it applicable for use in other domains. In this subsection, we describe our new use of VarCORE$^+$ on axTLS, an open-source highly configurable software system.

There is no requirement specification document for axTLS. For the evaluation, we thus synthesized the variability requirements (using the convention described in Table 6) manually from its KConfig model, which can be accessed via menuconfig and Config.in files in the code. Since VarCORE$^+$ currently only considers binary configuration options, 24 configuration options that require string and numerical inputs were excluded from the requirements/constraints synthesis, but will be added in future work. Thus, 74 variability requirements were produced for input to the RTW [48]. We applied the process as described in Section 3.2 to construct the RTW for the axTLS project. Table 11 shows an excerpt of the resulting axTLS RTW.

Table 11: Excerpts from the axTLS Requirements Traceability Worksheet (RTW)[48]

Requirements Traceability Worksheet (RTW)						
Req ID	**Requirements Specification**	**Valid**	**Rule**	**Parent Feature**	**Children Features**	**Source of Requirements**
AXTLS-4.1	The SSL library shall select SSL mode	1	R2	SSL_Library	SSL_Mode	synthesized from axTLS-2.1.5 menuconfig
AXTLS-4.1.1	The SSL mode shall use one and only one of the Server Only, the Server Only with Verify, the Server and Client, the Server and Client with Full Diagnostic or the Skeleton mode	1	R4	SSL_Mode	Server_Only, Server_Only_Verify, Server_Client, Server_Client_Full_Diagnostic, Skeleton_Mode	synthesized from axTLS-2.1.5 menuconfig
AXTLS-7	The axTLS may select the language bindings	1	R3	axTLS	Language_Bindings	synthesized from axTLS-2.1.5 menuconfig
AXTLS-7.3	The language bindings may use the Java bindings	1	R3	Language_Bindings	Java_Bindings	synthesized from axTLS-2.1.5 menuconfig
AXTLS-8	The axTLS may select the axSSL sample generation	1	R3	axTLS	axSSL_Sample_Generation	synthesized from axTLS-2.1.5 menuconfig
AXTLS-8.4	The axSSL sample generation may select the Jave language sample	1	R3	axSSL_Sample_Generation	Java_Language_Sample	synthesized from axTLS-2.1.5 menuconfig
AXTLS-24	The Java language sample requires the Java bindings	1	R7	Java_Language_Sample	Java_Bindings	synthesized from axTLS-2.1.5 Config.In

4.3.1 Static Analysis of Variability Requirements

Once the RTW was created, VarCORE$^+$ generated the variability model automatically. No unconnected features were detected during the variability model construction

for axTLS. Subsequent variability model analysis also showed that the generated model was valid and consistent. No dead feature or false-optional feature (an optional feature present in all products) was detected, as shown in Figs. 11b and 13. This result was expected as the variability requirements were synthesized from the implemented KConfig model.

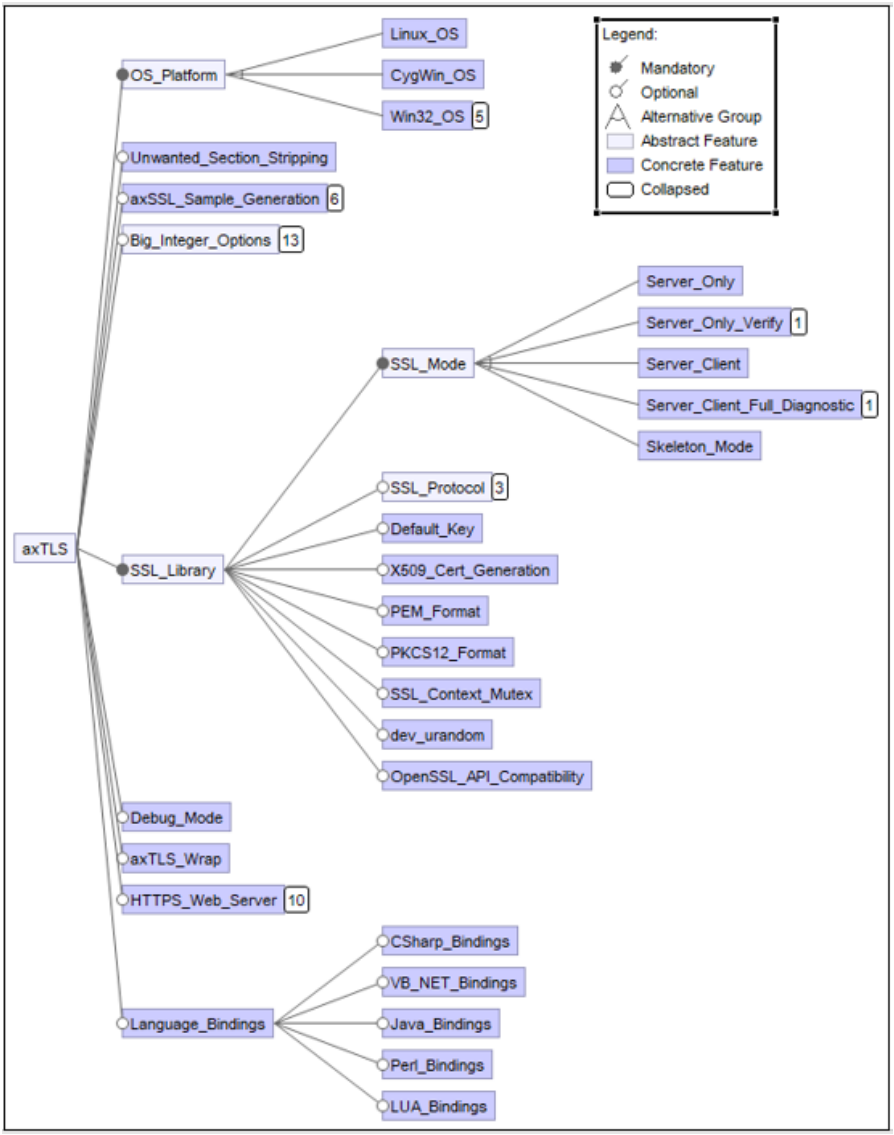


Fig. 13: axTLS Variability Model [49]. Some features were elided for display clarity

From the 74 variability requirements, we derived 71 Boolean configurable features for variability modeling. Based on the feature mapping between variability model and implementation (see Table 12), 9 of the 71 configurable features were set as abstract features, as they were not implemented in axTLS source code. See Fig. 13 for examples of abstract features. With 71 configurable features, axTLS has a large configuration space, and generating all valid configurations for variability analysis was not practical.

Table 12: Excerpts of Feature Map [50] between Variability Model and Code for axTLS

Features in Variability Model	**C Pre-processor Macros in axTLS (Code)**
Linux_OS	CONFIG_PLATFORM_LINUX
CygWin_OS	CONFIG_PLATFORM_CYGWIN
Win32_OS	CONFIG_PLATFORM_WIN32
Server_Only	CONFIG_SSL_SERVER_ONLY
Server_Only_Verify	CONFIG_SSL_CERT_VERIFICATION
Server_Client	CONFIG_SSL_ENABLE_CLIENT
Server_Client_Full_Diagnostic	CONFIG_SSL_FULL_MODE
Skeleton_Mode	CONFIG_SSL_SKELETON_MODE
Language_Bindings	CONFIG_BINDINGS
CSharp_Bindings	CONFIG_CSHARP_BINDINGS
VB_NET_Bindings	CONFIG_VBNET_BINDINGS
Java_Bindings	CONFIG_JAVA_BINDINGS
axSSL_Sample_Generation	CONFIG_SAMPLES
Java_Language_Sample	CONFIG_JAVA_SAMPLES
LUA_Language_Sample	CONFIG_LUA_SAMPLES
LUA_Build_Install	CONFIG_HTTP_BUILD_LUA

4.3.2 Combinatorial Interaction Analysis

axTLS supports multiple operating system (OS) platforms such as Linux, CygWin (Unix-like environment on Microsoft Windows) and Win32. We fixed and excluded a few configuration options as described below so that we could evaluate axTLS on our setup environment, described in Section 4.1:

- Linux_OS = True
- CSharp_Bindings = False
- VB_NET_Bindings = False
- LUA_Bindings = False
- LUA_Build_Install = False

Basically, we fixed configurations to run Linux OS platform only and excluded C#, VB .NET and LUA languages.

We then used VarCORE$^+$ to produce a minimal set of pair-wise combinatorial interaction configurations. This resulted in a total of 28 configurations [51] being generated. With 71 Boolean configurable features, and taking 18 variability constraints and a few fixed configurations above into account, the ACTS tool calculated 7339

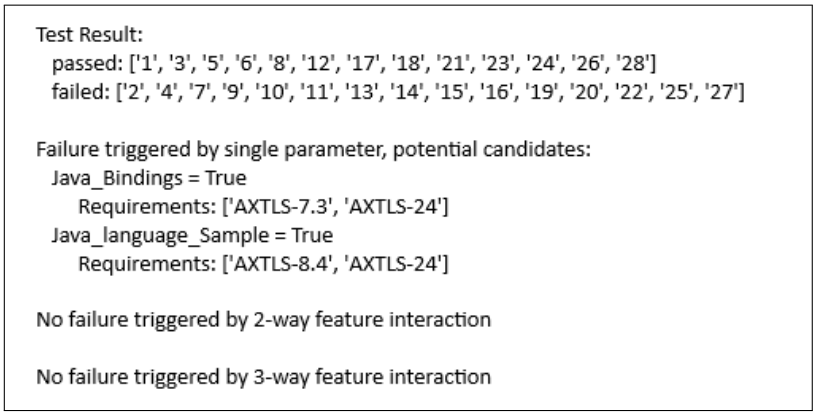


Fig. 14: axTLS Build Result and Analysis using VarCORE$^+$

possible pairs of feature combinations in axTLS and assured that all were covered at least once by the generated 28 configurations.

Subsequently, VarCORE$^+$ converted the 28 combinatorial configurations into corresponding code configuration files recognized by axTLS's build system. For example, the feature setting of *Linux_OS=true* in the VarCORE$^+$'s generated configurations was matched and transformed to *CONFIG_PLATFORM_LINUX=y* and *#define CONFIG_PLATFORM_LINUX 1* in axTLS's .config and config.h files, respectively (see Table 12 for the mapping). After conversion, VarCORE$^+$ built the axTLS's 28 product variants sequentially, one product variant per configuration file.

Fig. 14 shows the build result for each configuration file. 15 of the 28 configurations failed to build successfully. A useful capability of VarCORE$^+$ is that its automated identification of potential faulty-feature setting(s) after collecting the build result. VarCORE$^+$ analysis found two single-parameter faults (see Fig. 14) that contributed to the 15 build failures. Whenever *Java_Bindings*=True or *Java_Language_Sample*=True was configured in any configuration file, the axTLS compilation failed. VarCORE$^+$ then proceeded with its feature interaction analysis, but found no failure triggered by a 2-way or 3-way feature interaction.

Fig. 15 explains the root cause of the compilation error. When a new parameter: *SSL_EXTENSIONS** was added to the *ssl_client_new()* function in the axTLS-2.1.0 release, the two Java-bindings functions and Makefile, shown in the figure, were not updated to reflect the addition of the new parameter. Consequently, this variability bug was hidden until either configuration option *Java_Bindings*=True or *Java_Language_Sample*=True was configured for compilation.

VarCORE$^+$ also traced that these feature settings were specified by the requirements AXTLS-7.3, AXTLS-8.4, and constraint AXTLS-24 (see Table 11 for details). Although not found in axTLS, we note that having the variability requirements traceability available can be especially useful for troubleshooting a software configuration misuse problems (i.e., a configuration that should never be allowed). This type of issue is usually caused by a variability requirement specification error rather than by an implementation bug.

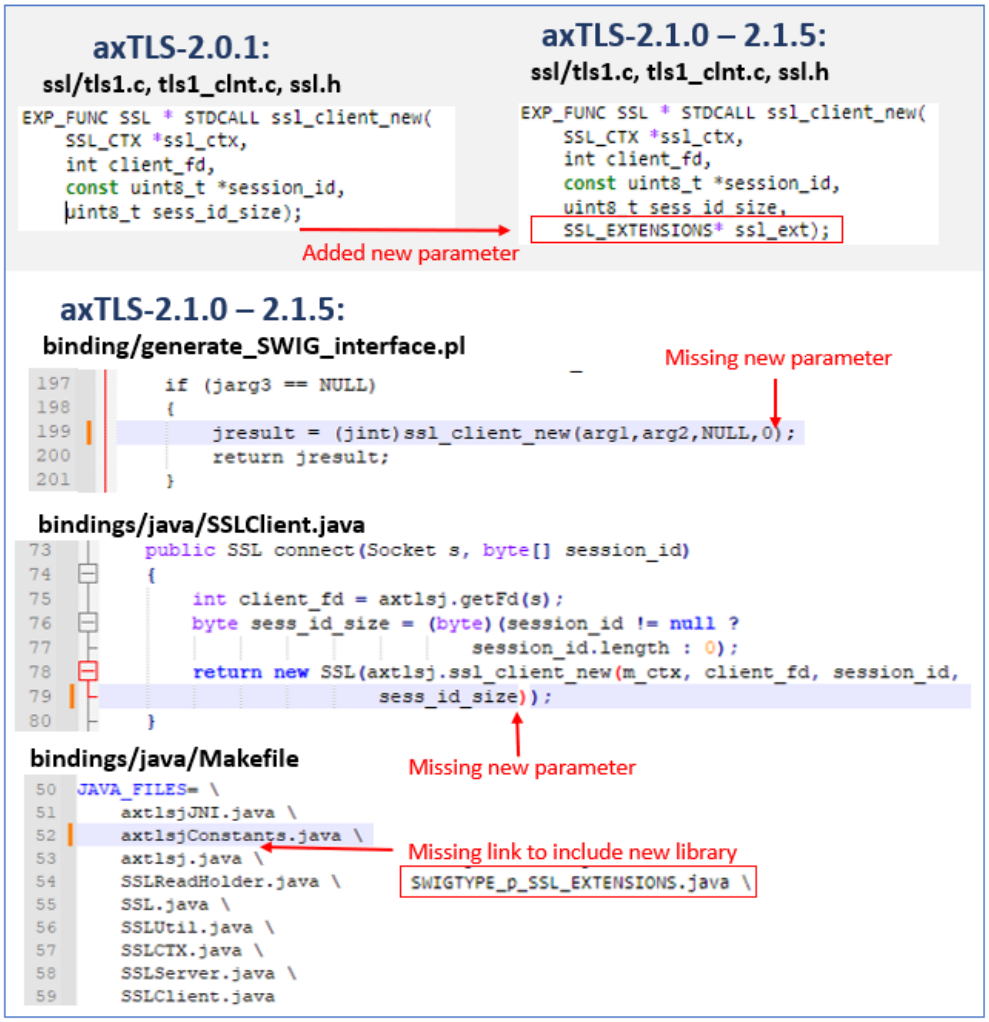


Fig. 15: axTLS Build Error Root Cause

4.3.3 Feature Interaction Analysis with Error Injection

In both cFE TIME and axTLS, VarCORE$^+$ discovered only single-parameter faults. To evaluate the effectiveness of VarCORE$^+$ in identifying inconsistency triggered by feature interaction, we fixed the Java Bindings issue we discovered and un-did a fix referred to as 'Header compile issue when "Create Language Bindings" is used' in the axTLS-2.1.5 change log[11]. That fix addressed a variability bug[12] triggered by feature interaction and was found by the Kmax tool [30].

[11] https://axtls.sourceforge.net/README/index.html
[12] https://github.com/paulgazz/kmax/blob/master/docs/bugs_found/2017-12-26_axtls_language_bindings.txt

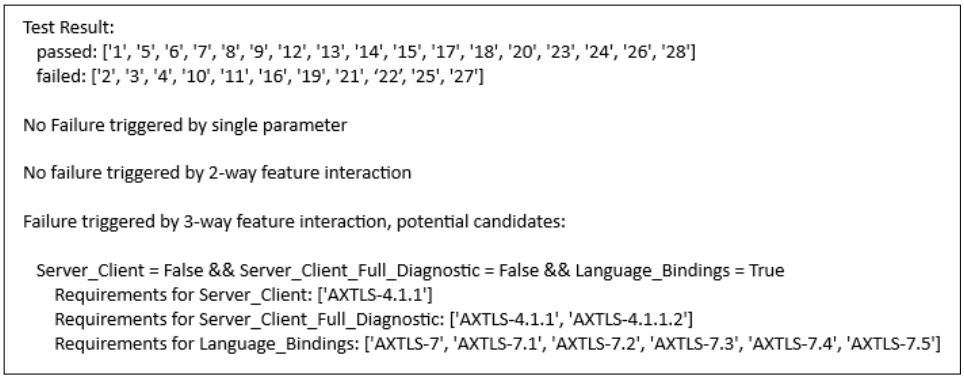


Fig. 16: axTLS Build Result and Analysis using VarCORE$^+$ with a 3-Way Feature Interaction Error Injected

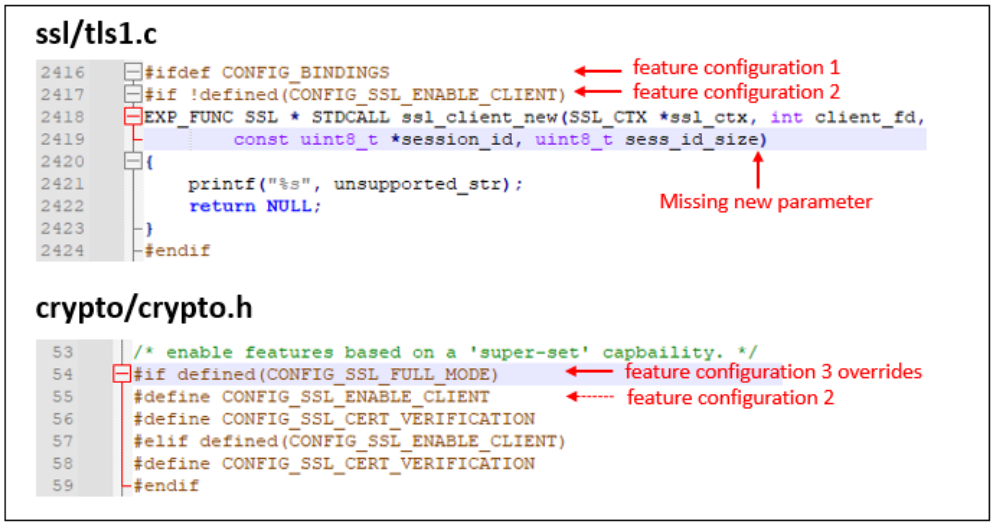


Fig. 17: Root Cause of the Build Failure Triggered by 3-Way Feature Interaction

We re-started the combinatorial interaction analysis using VarCORE$^+$ on the modified axTLS. Fig. 16 shows the new build result and analysis. With the modified axTLS, 10 out of 28 configurations failed to build. No single-parameter fault (as former discovered faults were fixed) and no 2-way feature interaction failures were detected. However, VarCORE$^+$ found that the 10 build failures were triggered by a 3-way feature interaction when *Language_Bindings*=True, *Server_Client*=False and *Server_Client_Full_Diagnostic*=False.

Fig. 17 illustrates the root cause of the variability bug triggered by this 3-way feature interaction. In ssl/tls1.c file, *ssl_client_new()* function (at line 2418 and 2419) would be called when *CONFIG_BINDINGS* was defined (see line 2416) and

CONFIG_SSL_ENABLE_CLIENT was not defined (see line 2417). Both code variables *CONFIG_BINDINGS* and *CONFIG_SSL_ENABLE_CLIENT* are mapped to features *Language_Bindings* and *Server_Client*, respectively (see Table 12). The third feature-configuration option was *CONFIG_SSL_FULL_MODE* (mapped to feature *Server_Client_Full_Diagnostic*) at crypto/crypto.h (line 54); it would override *CONFIG_SSL_ENABLE_CLIENT* configuration when it was defined. This verified that *ssl_client_new()* function would only be called when (1) *CONFIG_BINDINGS* was defined (i.e. set to True), (2) *CONFIG_SSL_ENABLE_CLIENT* was not defined (i.e., set to False) and (3) *CONFIG_SSL_FULL_MODE* was not defined (i.e., set to False). This aligned with the finding in Fig. 16. However, *ssl_client_new()* function was missing a new parameter, *SSL_EXTENSIONS**, and caused a compilation error. This is the same root cause as described in Fig. 15. Thus, when axTLS-2.1.5 fixed this issue triggered by 3-way feature interaction, it missed the variability bug described in Fig. 15 and found by VarCORE$^+$.

4.4 Research Questions

We now discuss what the results from our evaluation of VarCORE$^+$ on the two configurable systems indicate for our four research questions.

RQ1: Did VarCORE$^+$ find missing variability requirement(s)?

VarCORE$^+$ identified missing variability requirements in both applications. VarCORE$^+$ extracts all configurable features from variability requirements in the RTW and uses them as feature nodes to build a feature tree for connectivity analysis. Any discovery of unconnected feature (assume feature names are consistent in requirements) indicates that there is a missing requirement to specify the linkage between the unconnected feature and the full connected feature tree.

For example, in the evaluation for cFE TIME, VarCORE$^+$ detected that feature *Single_Processor* was not defined as it was an unconnected feature from the cFE TIME feature tree. As shown in VarCORE$^+$'s requirement traceability, feature *Single_Processor* was derived from variability constraint TIME-5.2. However, there was no variability requirement in the RTW to define or specify the hierarchical relationship between *Single_Processor* and any feature in the feature tree. Therefore, we can conclude that there was a missing requirement in the RTW.

For verification, the generated variability model diagram by VarCORE$^+$ can be used as a visual aid to check for the missing *Single_Processor* feature. The visual variability model diagram is useful as it allows the user to view all the features and their hierarchical relationships and constraints. It is easy to identify if a feature is missing from the diagram, which may indicate a missing requirement for the system.

In another example, axTLS's *changes, notes and errata*[13] show that some configuration options such as Java and Perl Bindings do not work for 64-bit Linux machine. However, there is no explicit variability feature for specifying 32-bit or 64-bit system, as shown in axTLS variability model diagram. This finding can prompt the developer to add the missing requirement.

[13] https://axtls.sourceforge.net/README/index.html

In summary, VarCORE$^+$ finds missing variability requirements using its feature-tree connectivity analysis. The variability model diagram is also useful for visually checking for any missing feature which may point to a missing requirement.

RQ2: Did VarCORE$^+$ find inconsistencies among the specified variability requirements?

Our results show that finding inconsistencies among the specified variability requirements and/or constraints is one of the major benefits of using VarCORE$^+$. Basically, VarCORE$^+$ transforms all the textual variability requirements into a variability model, so that it can perform variability model analysis to find inconsistency for repair.

For instance, the variability model analysis (see Section 4.2.1) for the flight software reported *Virtual_MET* as a false-optional feature (an optional feature present in all products) in the initial constructed variability model. This was caused by a conflict between the variability requirements TIME-6.1 and TIME-6.2. We showed in Section 4.2.1 that TIME-6.2 was erroneous. VarCORE$^+$ assisted by tracing TIME-6.2 to its original constraint statement ("TIME servers must be configured as using a virtual MET"), which can then be updated accordingly.

For axTLS, VarCORE$^+$ did not find any inconsistency among the specified variability requirements. This indicated that all the variability requirements were valid and consistent. This outcome was expected as the requirements were synthesized from a valid KConfig model of axTLS.

RQ3: Did VarCORE$^+$ find variability requirement(s) not implemented in the code?

We investigated this question and answered it affirmatively by running VarCORE$^+$'s combinatorial interaction analysis (see Section 4.2.2 and 4.3.2). The results for the flight software (cFE TIME) show that neither requirement TIME-6 nor requirement TIME-8 is implemented. The discovered configuration implementation bugs are known open issues tracked by cFS's Bug ID #109 (https://github.com/nasa/cFE/issues/109). Additionally, we analyzed the history of resolved bugs for cFE TIME and found a variability bug #2072 (https://github.com/nasa/cFE/issues/2072) which occurred only when feature *External_Time_Source* was configured to use feature *Spacecraft_Time*. We reproduced the bug (temporarily undoing the fix) when building the specific product variant and confirmed that the variability requirement was correct but not implemented correctly in the code. It is worth noting that this resolved bug could have been detected earlier if VarCORE$^+$ had been integrated into the cFS build system.

On the other hand, the result for axTLS shows that the discovered variability bug (see Fig. 15, let's call this *bugA*) will cause axTLS compilation failure whenever the single configuration option: *Java_Bindings* or *Java_Language_Sample* is enabled. Additionally, we reproduced the variability bug (see Fig. 17) which was fixed in axTLS-2.1.5 release by temporarily undoing the fix (let's call this *bugB*). *bugB* was introduced by a feature interaction. VarCORE$^+$ found that the compilation failure caused by *bugB* was triggered by a 3-way feature interaction by *Language_Bindings*, *Server_Client* and *Server_Client_Full_Diagnostic*. Later, we found that the root causes for *bugA* and *bugB*

are the same. However axTLS-2.1.5 fixed *bugB*, but missed *bugA*. Both issues could be discovered and fixed together if VarCORE$^+$ is integrated for use.

RQ4: Did VarCORE$^+$ scale to support configurable software in both domains?

Our evaluation on cFE TIME and axTLS shows that VarCORE$^+$ supports both of our two configurable software applications, which differ in domain, number of configurable features and build tools.

In VarCORE$^+$ the processes of (1) variability requirements collection into RTW, (2) variability model construction, and (3) static analysis of variability requirements are the same for both cFE TIME and axTLS. For other configurable software, our approach scales although it may be bounded by system memory and computational limits. However, VarCORE$^+$'s semi-automated, combinatorial interaction analysis must be modified and automated once to support a new project's specific build system and possibly new configuration framework. A project wanting to use VarCORE$^+$ thus incurs a relatively low, one-time overhead (as detailed in Section 3.5) in interfacing to VarCORE$^+$'s full functionality.

5 Discussion

It is known that the gap between requirements specification and implementation tends to grow if requirements updates are not performed in a timely manner [52]. To mitigate this, a developer can use VarCORE$^+$ after making a change to quickly check all valid configurations again. Moreover, the visualization provided by the variability model guides any slicing that an analyst wants to do to restrict the scope while investigating the requirements/code mismatch indicated by a failed build. VarCORE$^+$'s built-in focus on requirements traceability and visualization of relationships between features also eases the discovery and removal of problematic changes.

5.1 Threats to Validity

The internal validity (i.e., whether the measured use of VarCORE$^+$ to analyze the requirements accurately reflects the ability of the methodology to find errors) faced several challenges.

An internal threat to validity is that 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. Uncertainty modeling, as in [53], may offer a way to cope with uncertainties as to the extent to which the documented software requirements are adequate. Another threat to internal validity is that several steps in VarCORE$^+$ are manual (variability requirements/features extraction, features to code variables mapping) 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.

A significant threat to internal validity is that, although we assert that VarCORE$^+$ can save developer effort and increase developer understanding, we do not provide

evidence from human studies, with that work needed before adoption. Toward this, we have been in contact with NASA developers and hope to perform case studies in the future.

A threat to construct validity was the unavailability of the original axTLS requirements. This led us to synthesize them from its KConfig model. As a result, the fact that VarCORE$^+$ did not find any inconsistencies in axTLS's variability requirement (RQ2) was expected and not useful in evaluating RQ2. However, it did give us some confidence that the synthesized requirements correctly represented the KConfig model. Additionally, as a sanity check, we also compared our axTLS variability model with the feature model [54] that Knuppel et al. [55] translated from axTLS's KConfig model to the FeatureIDE file format to further assure the correctness of our synthesized requirements.

The external validity (i.e., the extent to which the conclusions asserted in this work can be generalized) are limited by our two applications and the primary domain of interest (spacecraft). However, we chose software systems from different domains, the second of which is highly configurable, to provide some test of its potential breadth of usage. Both are open-source projects with sizeable communities that have used them. Its use on other applications might yield different results. Thus, the evaluation reported here serves as a proof-of-concept study that would need to be repeated on other software systems to draw generalized expectations.

We provide a package with scripts so that other researchers and developers can replicate our results and use VarCORE$^+$ on their own projects. For applications with less variability and with requirements centralized in a single document, VarCORE$^+$ may not provide the same benefits. However, the relatively low manual overhead of VarCORE$^+$ suggests its use might still be beneficial in preventing requirements inconsistencies from propagating. Despite the threats, the evaluation of VarCORE$^+$ indicates advantages in reducing requirements errors prior to testing for configurable software systems.

5.2 Related Work

Variability models are well-studied and take multiple forms including feature models [41], tabular configurability models [7], and formal models [56, 57], created either from requirements or reverse engineered from code. Feature models are widely used to represent the commonality and variability of configurable systems [1, 11, 31, 37, 41].

Multiple approaches have been proposed to synthesize feature models from requirements and constraints in natural language (NL). Weston et al. [58] created a feature model from NL requirements using natural language processing (NLP) and a clustering algorithm, respectively. Davril et al. [59] mined the product descriptions from online software repositories such as SoftPedia and CNET to identify a set of features, then assembled them to construct a feature model. Mefteh et al. [60] derived a feature model from functional requirements of the product variants written in NL, mining features from product variants and constructing the feature model based on the hierarchical relationship and constraints among the features. These approaches have in common that they consist of two phases: automated extraction of requirements and features, which mainly relies on NLP techniques, and feature model construction.

However, even with major improvements in NLP techniques, it remains a challenging task to achieve high accuracy. It also requires significant, additional project investment in domain expertise to pre-process requirement specifications for model training, fine tune the models, and verify the results. Yu et al. [61], in a survey of transformation approaches between requirements and analysis models, reported restricted natural language, complicated pre-processing, and a great deal of user effort, with many approaches specific to use-case templates. Mavin and Wilkinson have provided five syntactic templates for expressing textual requirements that have been adopted by multiple organizations [62]. Although techniques are likely to continue improving rapidly, Zhao et al. [63] still recently noted a "huge discrepancy" between the state of the art and current NLP4RE research. Further, the variability requirements and constraints for many systems, including NASA's cFE TIME here, are scattered among documents without having fixed patterns or consistent formats. This also increases the difficulty of accurately automating requirements/constraints and feature extraction. For these reasons VarCORE$^+$ uses manual extraction of requirements/constraints from dispersed documents.

VarCORE$^+$ uses combinatorial interaction testing (CIT) techniques to generate product variants with t-way feature combinations from a feature model. Combinatorial interaction testing, described in Section 2.5, is a mature field. Hervieu et al. [64] introduced PACOGEN to generate pairwise test configurations from feature models using a constraint programming technique. PACOGEN was limited to pairwise (2-way) interactions between features and was not evaluated on real-world feature models. In contrast, VarCORE$^+$ uses the start-of-the-art CIT tool ACTS [43] to generate t-way combinatorial interactions among features, with t ranging from 1 to 6, and was evaluated on two real-world industrial applications, NASA's cFS and axTLS. Lopez-Herrejon et al. [65] proposed a framework for comparing CIT techniques on feature models.

Henand et al. [66] showed that the state-of-the-art CIT tools, including ACTS, which we use, cannot scale to systems with very large configurable features. As an example, for the Linux kernel with 6,888 configurable features and 343,944 constraints, the number of pair-wise and 3-way configurations could be more than 9.25E7 and 4.19E11. They proposed a randomized and a search-based approach that generates and prioritize configurations from feature model of very large configurable system within time and space budgets. However, the approach gains scalability by trading off test coverage, which may be problematic for critical systems. A recent study by Bombarda et al. [67], introduced a method to maximize the reuse of existing generated combinatorial configurations for testing evolving feature models. It used a greedy approach to compute the dissimilarity between test suites based on changes in the feature models to select re-usable test suites.

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" [68]. We hope that the VarCORE$^+$ framework may provide one such opportunity for joint innovation.

5.3 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 [69–71], and to enable variability code auto-analysis [29, 72] to further evaluate the consistency between requirements specification and implementation.

6 Conclusion

We proposed VarCORE$^+$, a new tool-supported framework to centralize, specify, and analyze variability requirements and constraints. Our approach uses variability-modeling and combinatorial interaction testing (CIT) techniques to perform its automated analyses and handle scalability issues. Results from our evaluation of VarCORE$^+$ on two diverse, configurable software systems show that it was effective at uncovering and helping resolve missing, inconsistent, and conflicting variability requirements and cross-tree constraints, including some that had been undetected even after implementation.

Acknowledgements We thank the anonymous reviewers for their insightful feedback to improve the paper. This work was supported in part by the National Science Foundation grant CCF-2211589.

References

[1] Batory, D.: Automated Software Design Volume 1, (2021). Lulu.com

[2] Pohl, K., Böckle, G., Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques, (2005). https://doi.org/10.1007/3-540-28901-1

[3] Nadi, S., Berger, T., Kästner, C., Czarnecki, K.: Where do configuration constraints stem from? an extraction approach and an empirical study. IEEE Trans. Software Eng. **41**(8), 820–841 (2015) https://doi.org/10.1109/TSE.2015.2415793

[4] Khor, C., Lutz, R.R.: Requirements analysis of variability constraints in a configurable flight software system. In: Schneider, K., Dalpiaz, F., Horkoff, J. (eds.) 31st IEEE International Requirements Engineering Conference, RE 2023, Hannover, Germany, September 4-8, 2023, pp. 244–254 (2023). https://doi.org/10.1109/RE57278.2023.00032

[5] Berger, T., Nadi, S.: Variability models in large-scale systems: A study and a reverse-engineering technique. In: Aßmann, U., Demuth, B., Spitta, T., Püschel, G., Kaiser, R. (eds.) Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany. LNI, vol. P-239, pp. 80–81 (2015)

[6] Oliveira, R.P., Mota Silveira Neto, P.A., Chen, Q., Almeida, E.S., Ahmed, I.: Different, really! A comparison of highly-configurable systems and single systems. Inf. Softw. Technol. **152**, 107035 (2022) https://doi.org/10.1016/J.INFSOF.2022.107035

[7] Cashman, M., Cohen, M.B., Ranjan, P., Cottingham, R.W.: Navigating the maze: the impact of configurability in bioinformatics software. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, pp. 757–767 (2018). https://doi.org/10.1145/3238147.3240466

[8] Apel, S., Atlee, J.M., Baresi, L., Zave, P.: Feature interactions: The next generation (Dagstuhl seminar 14281). Dagstuhl Reports **4**(7), 1–24 (2014) https://doi.org/10.4230/DAGREP.4.7.1

[9] Soares, L.R., Schobbens, P., Carmo Machado, I., Almeida, E.S.: Feature interaction in software product line engineering: A systematic mapping study. Inf. Softw. Technol. **98**, 44–58 (2018) https://doi.org/10.1016/J.INFSOF.2018.01.016

[10] Schmid, K., Rabiser, R., Becker, M., Botterweck, G., Galster, M., Groher, I., Weyns, D.: Bridging the gap: voices from industry and research on industrial relevance of SPLC. In: Mousavi, M.R., Schobbens, P. (eds.) SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A, pp. 184–189 (2021). https://doi.org/10.1145/3461001.3474301

[11] Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wasowski, A.: A survey of variability modeling in industrial practice. In: Gnesi, S., Collet, P., Schmid, K. (eds.) The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013, pp. 7–178 (2013). https://doi.org/10.1145/2430502.2430513

[12] Krüger, J., Çalikli, G., Berger, T., Leich, T., Saake, G.: Effects of explicit feature traceability on program comprehension. In: Dumas, M., Pfahl, D., Apel, S., Russo, A. (eds.) Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, pp. 338–349 (2019). https://doi.org/10.1145/3338906.3338968

[13] Ruiz, M., Hu, J.Y., Dalpiaz, F.: Why don't we trace? A study on the barriers to software traceability in practice. Requir. Eng. **28**(4), 619–637 (2023) https://doi.org/10.1007/S00766-023-00408-9

[14] Lutz, R.R., Mikulski, I.C.: Empirical analysis of safety-critical anomalies during operations. IEEE Trans. Software Eng. **30**(3), 172–180 (2004) https://doi.org/10.1109/TSE.2004.1271171

[15] Bayer, T.J.: Mars reconnaissance orbiter in-flight anomalies and lessons learned: An update. In: 2009 IEEE Aerospace Conference, pp. 1–11 (2009). https://doi.org/10.1109/AERO.2009.4839531

[16] Lutz, R.R.: Software engineering for space exploration. Computer **44**(10), 41–46 (2011) https://doi.org/10.1109/MC.2011.264

[17] Lutz, R.R.: Targeting safety-related errors during software requirements analysis. In: Notkin, D. (ed.) Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1993, Los Angeles, California, USA, December 7-10, 1993, pp. 99–106 (1993). https://doi.org/10.1145/256428.167069

[18] Kuhn, D.R., Kacker, R.N., Lei, Y.: Practical combinatorial testing. NIST special Publication **800**(142), 142 (2010)

[19] core Flight System: A paradigm shift in flight software development. https://cfs.gsfc.nasa.gov/

[20] Ngo, T.: NASA class A certification of core flight software (cFS). In: Flight Software Workshop (2021). https://www.youtube.com/watch?v=H7WuUxJeAsc&list=PLK-T7jljJ6zb0ALxjewsZ3amBTHPXH7qC&index=36&pp=iAQB

[21] 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]

[22] McComas, D., Wilmot, J., Cudmore, A.: The core flight system (cFS) community: Providing low cost solutions for small spacecraft. In: Annual AIAA/USU Conference on Small Satellites (2016)

[23] McComas, D.: OpenSatKit-a flight software system educational platform. In: Proceedings of the 35th AIAA/USU Conference on Small Satellites (2021)

[24] Ganesan, D., Lindvall, M., Ackermann, C., McComas, D., Bartholomew, M.: Verifying architectural design rules of the flight software product line. In: Muthig, D., McGregor, J.D. (eds.) Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings. ACM International Conference Proceeding Series, vol. 446, pp. 161–170 (2009). https://dl.acm.org/citation.cfm?id=1753258

[25] Ganesan, D., Lindvall, M., McComas, D., Bartholomew, M., Slegel, S., Medina, B., Krikhaar, R.L., Verhoef, C., Montgomery, L.P.: An analysis of unit tests of a flight software product line. Sci. Comput. Program. **78**(12), 2360–2380 (2013) https://doi.org/10.1016/J.SCICO.2012.02.006

[26] Miranda, D.J.F.: A 'new space' approach on spacecraft flight software development using NASA cFS framework. Master's thesis, INPE (2019)

[27] core Flight Executive Software Requirements Specification. https://github.com/nasa/cFE/blob/ee187426d08e0d4f0edf640d07381b8f676be8d3/docs/cfe%20requirements.docx (2017)

[28] axTLS Embedded SSL Project. https://axtls.sourceforge.net/

[29] Patterson, Z., Zhang, Z., Pappas, B., Wei, S., Gazzillo, P.: SugarC: Scalable desugaring of real-world preprocessor usage into pure C. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pp. 2056–2067 (2022). https://doi.org/10.1145/3510003.3512763

[30] Gazzillo, P.: Kmax: finding all configurations of kbuild makefiles statically. In: Bodden, E., Schäfer, W., Deursen, A., Zisman, A. (eds.) Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp. 279–290 (2017). https://doi.org/10.1145/3106237.3106283

[31] Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K.: A study of variability models and languages in the systems software domain. IEEE Trans. Software Eng. **39**(12), 1611–1640 (2013) https://doi.org/10.1109/TSE.2013.34

[32] Yilmaz, C., Fouché, S., Cohen, M.B., Porter, A.A., Demiröz, G., Koc, U.: Moving forward with combinatorial interaction testing. Computer **47**(2), 37–45 (2014) https://doi.org/10.1109/MC.2013.408

[33] Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43**(2), 11–11129 (2011) https://doi.org/10.1145/1883612.1883618

[34] Petke, J., Cohen, M.B., Harman, M., Yoo, S.: Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. IEEE Trans. Software Eng. **41**(9), 901–924 (2015) https://doi.org/10.1109/TSE.2015.2421279

[35] Overview of Rational DOORS. https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/doors/9.7.2?topic=engineering-requirements-management-doors-overview (2023)

[36] JIRA Overview. https://www.tutorialspoint.com/jira/jira_overview.htm

[37] Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines - Concepts and Implementation, (2013). https://doi.org/10.1007/978-3-642-37521-7

[38] Beuche, D., Dalgarno, M.: Software product line engineering with feature models.

Overload Journal **78**, 5–8 (2007)

[39] Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications, (2009). http://eu.wiley.com/WileyCDA/WileyTitle/productCd-EHEP000863.html

[40] Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., Czarnecki, K.: What is a feature?: a qualitative study of features in industrial software product lines. In: Schmidt, D.C. (ed.) Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, pp. 16–25 (2015). https://doi.org/10.1145/2791060.2791108

[41] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst (1990)

[42] Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G.: Mastering Software Variability with FeatureIDE, (2017). https://doi.org/10.1007/978-3-319-61443-4

[43] ACTS User Guide. https://csrc.nist.gov/csrc/media/Projects/automated-combinatorial-testing-for-software/documents/acts_user_guide_3.2.pdf

[44] Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empir. Softw. Eng. **16**(1), 61–102 (2011) https://doi.org/10.1007/S10664-010-9135-7

[45] Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J.: IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. Softw. Test. Verification Reliab. **18**(3), 125–148 (2008) https://doi.org/10.1002/STVR.381

[46] cFE Functional Requirement. https://github.com/nasa/cFE/blob/main/docs/cFE_FunctionalRequirements.csv

[47] cFE TIME Requirements Traceability Worksheet. https://github.com/chinkhor/VarCORE2/blob/main/artifacts/RTW_cfs.csv

[48] axTLS Requirements Traceability Worksheet. https://github.com/chinkhor/VarCORE2/blob/main/artifacts/RTW_axtls.csv

[49] axTLS Variability Model. https://github.com/chinkhor/VarCORE2/blob/main/artifacts/axTLSFeatureModel.xml

[50] Feature Map between Feature Model and Code for axTLS. https://github.com/chinkhor/VarCORE2/blob/main/axtls_map

[51] Pair-wise Combinatorial Interaction Configurations for axTLS. https://github.com/chinkhor/VarCORE2/blob/main/artifacts/axtls_ACTS_output2.csv

[52] Lethbridge, T., Singer, J., Forward, A.: How software engineers use documentation: The state of the practice. IEEE Softw. **20**(6), 35–39 (2003) https://doi.org/10.1109/MS.2003.1241364

[53] Cailliau, A., Lamsweerde, A.: Handling knowledge uncertainty in risk-based requirements engineering. In: 2015 IEEE 23rd International Requirements Engineering Conference (RE), pp. 106–115 (2015). https://doi.org/10.1109/RE.2015.7320413

[54] axTLS Feature Model. https://github.com/AlexanderKnueppel/is-there-a-mismatch/blob/master/Data/LargeFeatureModels/KConfig/axTLS.xml

[55] Knüppel, A., Thüm, T., Mennicke, S., Meinicke, J., Schaefer, I.: Is there a mismatch between real-world feature models and product-line research? In: Bodden, E., Schäfer, W., Deursen, A., Zisman, A. (eds.) Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp. 291–302 (2017). https://doi.org/10.1145/3106237.3106252

[56] Chechik, M., Stavropoulou, I., Disenfeld, C., Rubin, J.: FPH: efficient non-commutativity analysis of feature-based systems. In: Russo, A., Schürr, A. (eds.) Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10802, pp. 319–336 (2018). https://doi.org/10.1007/978-3-319-89363-1_18

[57] Mansoor, N., Saddler, J.A., Silva, B.V.R., Bagheri, H., Cohen, M.B., Farritor, S.: Modeling and testing a family of surgical robots: an experience report. In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (eds.) Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, pp. 785–790 (2018). https://doi.org/10.1145/3236024.3275534

[58] Weston, N., Chitchyan, R., Rashid, A.: A framework for constructing semantically composable feature models from natural language requirements. In: Muthig, D., McGregor, J.D. (eds.) Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings. ACM International Conference Proceeding Series, vol. 446, pp. 211–220 (2009). https://dl.acm.org/citation.cfm?id=1753265

[59] Davril, J., Delfosse, E., Hariri, N., Acher, M., Cleland-Huang, J., Heymans, P.: Feature model extraction from large collections of informal product descriptions. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian

Federation, August 18-26, 2013, pp. 290–300 (2013). https://doi.org/10.1145/2491411.2491455

[60] Mefteh, M., Bouassida, N., Ben-Abdallah, H.: Mining feature models from functional requirements. Comput. J. **59**(12), 1784–1804 (2016) https://doi.org/10.1093/COMJNL/BXW027

[61] Yue, T., Briand, L.C., Labiche, Y.: A systematic review of transformation approaches between user requirements and analysis models. Requir. Eng. **16**(2), 75–99 (2011) https://doi.org/10.1007/S00766-010-0111-Y

[62] Mavin, A., Wilkinson, P.: Ten years of EARS. IEEE Softw. **36**(5), 10–14 (2019) https://doi.org/10.1109/MS.2019.2921164

[63] Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K.J., Ajagbe, M.A., Chioasca, E., Batista-Navarro, R.T.: Natural language processing for requirements engineering: A systematic mapping study. ACM Comput. Surv. **54**(3), 55–15541 (2022) https://doi.org/10.1145/3444689

[64] Hervieu, A., Baudry, B., Gotlieb, A.: PACOGEN: automatic generation of pairwise test configurations from feature models. In: Dohi, T., Cukic, B. (eds.) IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011, pp. 120–129 (2011). https://doi.org/10.1109/ISSRE.2011.31

[65] Lopez-Herrejon, R.E., Ferrer, J., Chicano, F., Haslinger, E.N., Egyed, A., Alba, E.: Towards a benchmark and a comparison framework for combinatorial interaction testing of software product lines, vol. abs/1401.5367 (2014). http://arxiv.org/abs/1401.5367

[66] Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Traon, Y.L.: Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Trans. Software Eng. **40**(7), 650–670 (2014) https://doi.org/10.1109/TSE.2014.2327020

[67] Bombarda, A., Bonfanti, S., Gargantini, A.: On the reuse of existing configurations for testing evolving feature models. In: Arcaini, P., Beek, M.H., Perrouin, G., Reinhartz-Berger, I., Machado, I., Vergilio, S.R., Rabiser, R., Yue, T., Devroey, X., Pinto, M., Washizaki, H. (eds.) Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B, SPLC 2023, Tokyo, Japan, 28 August 2023- 1 September 2023, pp. 67–76 (2023). https://doi.org/10.1145/3579028.3609017

[68] Gazzillo, P., Cohen, M.B.: Bringing together configuration research: Towards a common ground. In: Scholliers, C., Singer, J. (eds.) Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022, Auckland, New Zealand,

December 8-10, 2022, pp. 259–269 (2022). https://doi.org/10.1145/3563835.3568737

[69] Abualhaija, S., Arora, C., Sabetzadeh, M., Briand, L.C., Vaz, E.: A machine learning-based approach for demarcating requirements in textual specifications. In: Damian, D.E., Perini, A., Lee, S. (eds.) 27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019, pp. 51–62 (2019). https://doi.org/10.1109/RE.2019.00017

[70] Nistala, P.V., Rajbhoj, A., Kulkarni, V., Soni, S., Nori, K.V., Reddy, R.: Towards digitalization of requirements: generating context-sensitive user stories from diverse specifications. Autom. Softw. Eng. **29**(1), 26 (2022) https://doi.org/10.1007/S10515-022-00324-2

[71] Zhang, J., Chen, S., Hua, J., Niu, N., Liu, C.: Automatic terminology extraction and ranking for feature modeling. In: 30th IEEE International Requirements Engineering Conference, RE 2022, Melbourne, Australia, August 15-19, 2022, pp. 51–63 (2022). https://doi.org/10.1109/RE54965.2022.00012

[72] Schubert, P.D., Gazzillo, P., Patterson, Z., Braha, J., Schiebel, F., Hermann, B., Wei, S., Bodden, E.: Static data-flow analysis for software product lines in C. Autom. Softw. Eng. **29**(1), 35 (2022) https://doi.org/10.1007/S10515-022-00333-1