Carl von Ossietzky
Universität
Oldenburg

Fakultät II
Department für Wirtschaftsinformatik
Intelligente Transportsysteme

M.Sc. Wirtschaftsinformatik

Masterarbeit

Feature-orientierte Entwicklung von Raumfahrzeugen

Erstgutachter: Prof. Dr. Frank Köster
Zweitgutachter/Betreuer: Dr. Philipp Chrszon

Vorgelegt von:
Jendrik Mann,

Abgabe:
Oldenburg, 22.04.2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

AOCS = Attitude and Orbit Control System
CDM = Conceptual Data Model
CE = Concurrent Engineering
DLR = Deutsches Zentrum für Luft- und Raumfahrt
ED = Element Definition
EPS = Energy Production System
FOSD = Feature-orienteds Software Development
FT = Feature Tree
GSEL = Generic System Engineering Language
MBSE = Model-based System Engineering
OBC = On-Board-Computer
OBD = On-Board-Diagnostics
PT = Product Tree
PTD = Product Tree Domain
RW = Reaction Wheel
SM = System Model
SRS = Software Requirements Specification
TCS = Temperature Control System
UI = User Interface

Glossar

Ad-hoc Produktlinie Eine Produktlinie, die im Gegenteil zu den herkömmlichen Produktlinien häufig kurzfristig manipuliert wird (Eigenkreation) . VII

Concept Erweiterung für Virtual Satellite basierend auf der GSEL . VII

Configuration Tree Zweite Phase des Entwurfsprozesses in Virtual Satellite, in der eine Konfiguration erstellt wird . VII

Domänenartefakt Wiederverwendbare Artefakte aus Domain-Engineering-Prozess [13]. VII

Feature Funktionale Abstraktionseinheit eines Systems . VII

Feature Tree Schnittstelle zwischen Product Tree, Feature-Modell und Configuration Tree . VII

Feature-Modell Ein Feature-Set, das Beziehungen zwischen den Features definiert . VII

Product Tree Erste Phase des Entwurfsprozesses in Virtual Satellite, in der die Komponenten definiert werden . VII

SysML Modellierungssprache des MBSE . VII

Variante Repräsentation eines variablen Objekts innerhalb eines Domänenartefakts [13]. VII

Variationspunkt/Entscheidungspunkt Repräsentation eines Variabilitätssubjekts innerhalb eines Domänenartefakts, welche durch kontextuelle Informationen angereichert ist [13]. VII

Virtual Satellite MBSE-Software des DLR. VII

Zusammenfassung

Diese Masterarbeit nutzt das Konzept einer Software-Produktlinie aus der Feature-orientierten Softwareentwicklung, um den iterativen Satellitenentwurfsprozess der Software Virtual Satellite zu verbessern. Alle Varianten, die während des Entwurfsprozesses verfallen würden, werden in einem Feature-Modell dokumentiert und späterer Zugriff ermöglicht. Es wird bewiesen, dass die bestehende Baumstruktur von Virtual Satellite genutzt werden kann, sodass der Feature Tree als Bindeglied zwischen Product Tree und Configuration Trees fungieren und die Virtual Satellite-Elemente in ein Feature-Modell und wieder zurück übersetzen kann. Alle validen Varianten aus dem Entwurfsprozess werden gefunden und können als Configuration Tree instanziiert werden.

1 Einleitung

Die Anzahl der Satelliten im All nimmt rasant zu, vor allem durch das Aufkommen von Mikro- und Minisatelliten, die auch kleineren Institutionen den Zugang zur Raumfahrt ermöglichen [10]. Der Satellitenentwurfsprozess ist dabei immer ein entscheidender Faktor, der über die Kosten und Dauer eines Raumfahrtprojekts entscheidet. Er ist unterteilt in die Phasen A bis F, die in Gesamtheit über zehn Jahre andauern können. Phase A fokussiert die Machbarkeit der Mission, dann folgen detaillierte Missionsplanung, zwei Phasen für Design, Entwicklung, Herstellung, Integration und Verifikation, anschließend die Durchführung und Analyse und zuletzt Phase F, in der die Entsorgung des Raumfahrzeugs durchgeführt wird. Das Systems Engineering unterstützt bei diesen Phasen in den Aufgabenbereichen Anforderungserhebung, Design und Konfiguration der physischen Architektur und Verifikation. Dabei sollen alle Stakeholder und Teammitglieder in alle Projektphasen integriert werden. Um den hohen Kosten und einer langen Projektzeit entgegenzuwirken und die Qualität des Produkts zu erhöhen, wurden Concurrent Engineering (CE) eingeführt. CE umfasst Techniken, mit denen Entwurf, Entwicklung, Beschaffung und Herstellung in Teamarbeit stattfindet, die nahezu in Echtzeit agiert. Dazu werden Tools über alle Disziplinen hinweg integriert und sich häufig auch die selbe Concurrent Engineering Facility geteilt [9]. Model-based System Engineering (MBSE) ist ein weiteres hilfreiches Tool im System Engineering, mit dem die vielen Dokumente im Entwurfsprozess gegen ein System Model getauscht werden, an dem alle Stakeholder und Teammitglieder gleichzeitig arbeiten, Informationen einpflegen und einholen können. Alle sind sich in Modellsprache, Informationsinhalt und -struktur einig [14]. Virtual Satellite ist ein vom Deutschen Zentrum für Luft- und Raumfahrt (DLR) entwickeltes MBSE-Tool, das die gemeinsame Arbeit an einem Satellitenentwurf ermöglicht. Momentan läuft der Entwicklungsprozess eines Raumfahrzeuges mit Virtual Satellite sehr iterativ ab. Es wird eine Variante erstellt und mit den Anforderungen abgeglichen. Sollte dann etwas geändert werden, verfällt die zuletzt erstellte Variante. Doch während des Entwicklungsprozesses eines Satelliten kommt der Entwickler häufig an einen Entscheidungspunkt, an dem er zwischen mehreren Optionen auswählen muss und die nicht gewählte Variante direkt verfällt. Dabei hat er eigens aufgestellte Constraints, die er berücksichtigen muss oder die bisherige Konfiguration verbietet die Auswahl mancher Optionen. Um diesen Prozess zu automatisieren und die Möglichkeit zu bieten, sich nicht direkt entscheiden zu müssen, soll eine Ad-hoc-Produktlinie den Konfigurationsprozess verfolgen, mögliche Varianten anbieten und unmögliche Varianten ausblenden. Dazu soll das Konzept eines Features eine

Entwurfsentscheidung repräsentieren, die sich durch ihre Parameter von anderen Entwurfsentscheidungen unterscheidet und ggf. auch Constraints implementiert, die die Verwendung eines anderen Features verbietet. Ein Ingenieur kann z.B. vor der Entscheidung stehen, fest installierte Solarpaneele oder Faltpaneele zu verwenden. Diese Entscheidung kann entweder automatisiert abgenommen werden, weil ein Constraint besagt, dass z.B. ein gewisses Maximalgewicht mit Faltpaneelen überschritten werden würde oder sie kann auf später verschoben werden, weil z.B. die Information des Maximalgewichts noch nicht vorliegt. Im zweiten Fall würden die beiden Features jeweils in zwei unterschiedlichen Varianten der Ad-hoc-Produktlinie gespeichert werden. Im Laufe des Entwicklungsprozesses können weitere Features zu diesen Varianten hinzugefügt werden. Dabei kann es sein, dass eine Variante durch ein später hinzugefügtes Feature natürlicherweise nicht mehr möglich ist. Sollten aber beide Varianten noch möglich sein, kann der zur Entscheidung bereite Ingenieur eine Variante auswählen und damit weitermachen oder den Entwurfsprozess beenden. Eine Begutachtung der Varianten und eine Variantenselektion kann dabei jederzeit durchgeführt werden.
Die Masterarbeit fokussiert dementsprechend den Problem Space des FOSD-Prozesses mit der Definition von Features und Feature-Modell. Im ersten Schritt der Abschlussarbeit muss dafür ermittelt werden, wie die FOSD-Konzepte aus dem Problem Space in den MBSE-Prozess von Virtual Satellite integriert werden können. Dazu gehört die Entwicklung eines Konzepts, wie eine Ad-hoc-Produktlinie den Entwurfsprozess und die möglichen Varianten verfolgen kann. Der Produktlinie liegt ein Feature-Modell zugrunde, für das beantwortet werden muss, an welcher Stelle es definiert wird und wie es dynamisch verändert werden kann, um den Entwurfsprozess abzubilden. Weiterhin muss beantwortet werden, wann und wie die Features entstehen. Werden sie durch den Ingenieur definiert oder entstehen sie automatisch, sobald ein Entscheidungspunkt erreicht wird? Sobald die Features definiert wurden, muss ermittelt werden, wie sie in das Feature-Modell eingepflegt werden. In der programmiertechnischen Phase werden die erforschten Konzepte dann prototypisch implementiert. Hier wird ein Eclipse-basiertes Plug-In für Virtual Satellite entworfen, das in eine oder mehrere der Entwicklungsphasen eingreift, eine Produktlinie aufbaut und dem Nutzer verschiedene Varianten anbietet. In der Evaluation soll praktisch erprobt werden, ob das System zum Entwurfsprozess passende Varianten anbietet, korrekt Features erstellt und in das Feature-Modell einbindet.

1.1 Forschungsfragen

1. Wie lassen sich FOSD-Konzepte des Problem Space in den MBSE-Prozess von Virtual Satellite integrieren, sodass der Nutzer korrekte Varianten vorgeschlagen bekommt?

2. Wie kann eine dynamische Produktlinie den Entwurfsprozess samt möglicher Varianten verfolgen?

   (a) Wie sieht das zugrundeliegende Feature-Modell aus?

   (b) An welcher Stelle wird das Feature-Modell definiert?

   (c) Wie kann das Feature-Modell dynamisch angepasst werden?

3. Wie sehen die Features aus?

   (a) Welche Elemente des Raumfahrzeugentwurfs repräsentieren sie?

   (b) An welcher Stelle des Entwurfs werden sie definiert?

4. Wie können die Anforderungen und Constraints des Nutzers die Produktlinie beeinflussen?

5. Wie läuft der Prozess der Variantenselektion konzeptionell, im Code und für den Nutzer ab?

6. Wie wird mit Anforderungen umgegangen (d.h. es existiert keine mögliche Variante), die nicht erfüllt werden können?

2 Verwandte Arbeiten

In diesem Kapitel werden die Begriffe Model-based System Engineering und Feature-oriented Software Development erklärt sowie eine Einführung in die Software Virtual Satellite geliefert.

2.1 Model-based System Engineering

System Engineering hatte jahrelang (Papier-) Dokumente als Basis, um den Entwicklungsprozess zu dokumentieren und gemeinsam am Entwurf zu arbeiten. Doch ein Raumfahrtsystem besteht aus vielen Subsystemen, die jeweils wieder von einzelnen Teams entwickelt werden. So ist es schwer, die ganze Entwicklungsphase teamübergreifend konsistent zu arbeiten, wenn permanent textbasierte Dokumente verändert und ausgetauscht werden müssen. MBSE wurde daraufhin entwickelt, um die Dokumente mit einem konsistenten Modell auszutauschen, das alle Systeme und Subsysteme übersichtlich in Tabellen, Diagrammen und Zahlen beschreibt. Mit dieser Struktur können sowohl Menschen als auch Computer effektiver arbeiten und die Verifikation, Analyse und Wiederverwendung der Modelle wird erleichtert. Die Kontrolle über Dokumente wird gegen die Kontrolle über das Modell getauscht. Die European Space Agency hat sich zum Ziel gesetzt, die MBSE Prozesse der größten europäischen Raumfahrtintegratoren zusammen- und einen Industriestandard einzuführen [1]. Das Hauptartefakt des MBSE ist das System Model, das meist in einer Modellierungssprache wie SysML oder UML entwickelt und in einem Repository gespeichert wird. Das Repository sorgt für die Kongruenz in den System Models auf die die Stakeholder und Teams zugreifen. Die wichtigsten Elemente des Systems, d.h. also Anforderungen, Struktur, Verhalten und Parameter werden durch das System Model repräsentiert. Diese gemeinsame Wissensbasis verbessert die Kommunikation zwischen den Stakeholdern und den Teammitgliedern, indem es eine gemeinsame Wahrnehmung ein gleiches Verständnis der Domäne erzwingt. Die Präzision und die Integrität des Entwurfs nimmt durch die Vermeidung von eigenen, unterschiedlichen Informationsrepräsentationen zu. Informationen werden verfolgbar und Artefakte können wiederverwendet werden. All diese Vorteile haben ein geringeres Risiko in der Entwicklung des Systems zur Folge. In einer Welt, die Menschen aus allen Ländern, Sprachen und Kulturen in der Arbeit an Systems Engineering Projekten vereint, ist es unvermeidlich, dass innerhalb des Projekts alle dieselbe Sprache sprechen. Das macht MBSE zu einer wichtigen Herangehensweise des System Engineerings [14]. Die ESA hat bereits ein ausschließlich für interne Missionen genutztes MBSE-Tool namens Open Concurrent Design Tool (OCDT) entwickelt. Der Client ist ein Add-in für Microsoft Excel. Der Server bietet eine PostgreSQL-Datenbank zur Speicherung der Daten und Modell. Es soll mehr als 20 Nutzer unterstützen und deren Modelle zweimal die Minute synchronisieren ([7]). Leider wurde mir auf meine Nachfrage kein Zugriff gewährt. Im MBSE-Bereich existiert außerdem eine Arbeit, die ähnlich zu dieser Arbeit eine Äutomatisierte Ableitung von Varianten aus SysML-basierten Systemarchitekturen"[16] erforscht hat. Die Ergebnisse der Arbeit sind vielversprechend, sie bezieht sich aber auf die Sprache SysML. Die folgende Ausarbeitung soll sich hingegen auf die Architektur von Virtual Satellite und Varianten von Raumfahrzeugen spezialisieren.

2.1.1 Virtual Satellite

Virtual Satellite wurde in enger Zusammenarbeit mit der Concurrent Engineering Facility des DLR in Bremen entwickelt und setzt den Fokus auf den Austausch von Informationen des Entwurfs über den Projektlebenszyklus hinweg. Dabei folgt Virtual Satellite dem Konzept des MBSE und bietet eine System-Engineering-Datenbank an, die den Satellitenentwurfsprozess unterstützt. Etablierte Entwurfsprozesse und -aufgaben sind mit dieser Datenbank verbunden und verteilen darüber ihre Informationen an alle Stakeholder. Durch diese gemeinsame Informationsbasis, die allen Stakeholdern als Grundlage dient, entsteht eine "Single Source of Truth", wodurch Fehler aus Differenzen im Informationsstand vermieden werden. Damit die Stakeholder die Informationen uniformiert erstellen und bearbeiten können, bedarf es einer Semantik, die von der Plattform vorgegeben und von allen benutzt werden muss. Das *Conceptual Data Model* stellt die Bausteine für die Informationserstellung zur Verfügung und erzwingt somit die semantische Korrektheit. In Abb. 1 ist die Beziehung

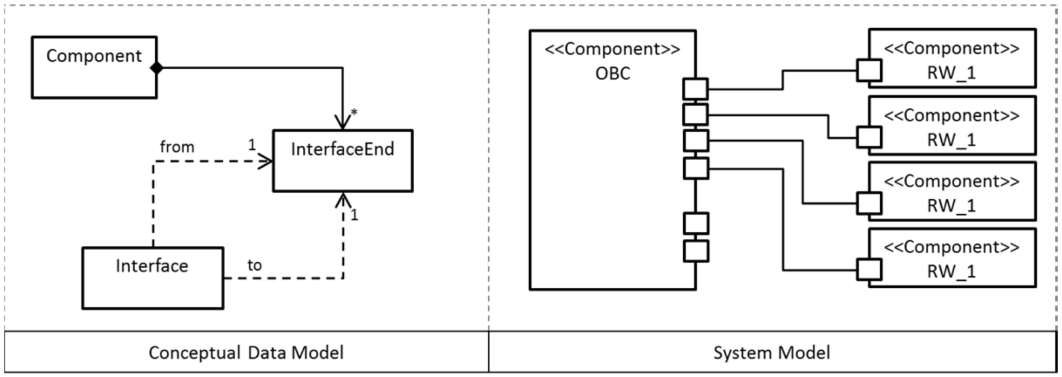


Abbildung 1: *System Model* und *Conceptual Data Model* [8]

eines *System Model* zu seinem *Conceptual Data Model* (CDM) in UML/SysML-Notation dargestellt. Das CDM zeigt eine Komponente, die ein oder mehrere *InterfaceEnds* haben kann. Außerdem zeigt es ein Interface, das als Input (*from*) und Output (*to*) auf ein *InterfaceEnd* verweist. Das SM hat eine Komponente *OBC* und vier Komponenten *RW_1*. Das *OBC* ist jeweils mit einem *Interface* mit einem *RW* verbunden. Das Interface mündet auf jeder Seite an das *InterfaceEnd* der jeweiligen Komponente [8]. Das CDM besteht aus der Generic System Engineering Language (GSEL) und den sog. *Concepts*, die ebenfalls mit der GSEL erstellt werden. Die GSEL wurde auf Basis des Eclipse Ecore implementiert und fungiert als Metamodell für das System Engineering. Einen detaillierten Einblick in die GSEL gibt das Kapitel 4.2. *Concepts* erweitern Virtual Satellite um Funktionalitäten und Strukturen. Da alles auf der GSEL aufbaut, können die Ingenieure am selben System Model arbeiten, obwohl sie verschiedene *Concepts* aktiviert haben. Wichtig für die nachstehende Implementierung ist das *Product Structure Concept*, das zusammen mit der Virtual Satellite Core Edition ([2]) als Grundlage genommen wird. Das *Product Structure Concept* unterteilt den Entwurfsprozess in 4 Phasen, die in Abb. 3 als Vererbungsstruktur dargestellt sind ([11]):

1. Im Product Tree definiert der Ingenieur das Equipment, das im Raumfahrzeug verwendet werden soll. Die Komponenten werden als Element Definition (ED) instanziiert

oder als Product Tree Domain (PTD), wenn sie eine Komponentengruppe beschreiben. Die EDs und PTDs können Attribute haben, die Parameter der Komponente darstellen.

2. Im Configuration Tree werden Instanzen vom Equipment erstellt, die der gewünschten Konfiguration entsprechen. Die Instanzen erben ihre Informationen vom Product Tree, können aber noch weitere instanzspezifische Informationen erhalten. Virtual Satellite Core ermöglicht im Gegensatz zu den anderen Editionen das Erzeugen mehrere Configuration Trees.

3. Im Assembly Tree werden Informationen zum Raumfahrzeug hinzugefügt. Alle Informationen aus dem Configuration Tree werden geerbt. Der Assembly Tree kann als eine Instanz der Raumfahrzeugkonfiguration aus dem Configuration Tree betrachtet werden.

4. Informationen aus dem Product Storage stammen vom realen Equipment, das u.U. noch im Lager aufzufinden ist und werden am Ende in den Assembly Tree integriert.

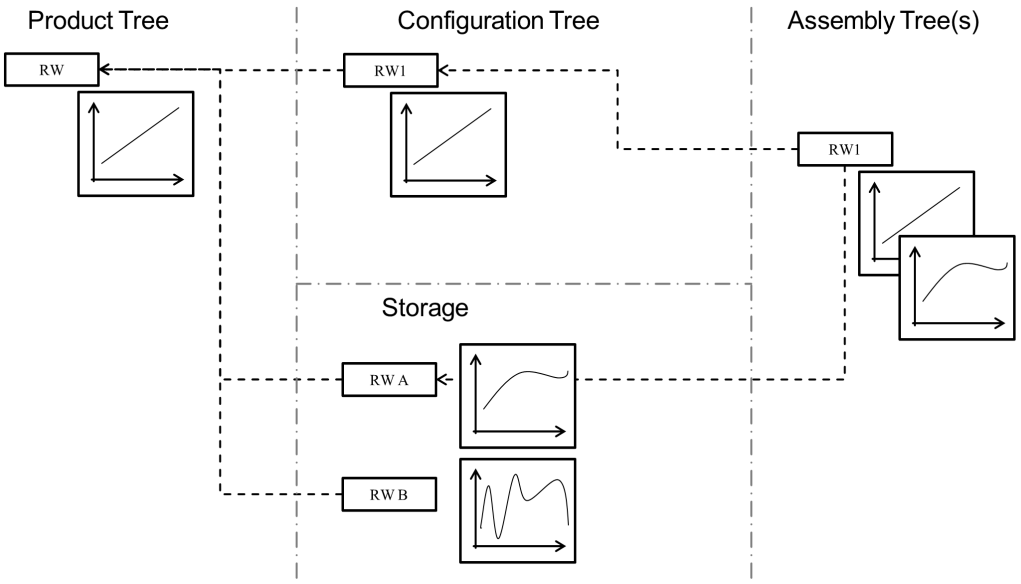


Abbildung 2: Vererbung innerhalb der Baumstruktur des *Product Structure Concepts* [11]

2.2 Feature-oriented Software Development

Feature-oriented Software Development ist ein Konzept zur stark skalierbaren Entwicklung angepasster Softwaresysteme. Die Idee ist, eine Software in ihre Bestandteile, die sog. Features zu zerlegen. Durch die Komposition der Features können dann auf die Bedürfnisse des Nutzers und des Anwendungsbereichs zugeschnittene Systeme erzeugt werden. Der Begriff "Feature" unterliegt einer Spannung zwischen der abstrakten und der Implementierungssicht. Features können zum einen als für den Nutzer sichtbare, deutlich abgrenzbare Funktionseinheit eines Softwaresystems gelten, aber zum anderen auch ein Inkrement einer Funktionalität des Systems sein, das es zu implementieren, testen, ausliefern und warten gilt. Das Set aus

Features wird auch als Software-Produktlinie bezeichnet. Aus ihr können viele verschiedene Produkte (auch Varianten oder Konfigurationen genannt) erstellt werden, die gemeinsame und unterschiedliche Features besitzen. Die Wiederverwendbarkeit der Bestandteile steht bei dem Entwurf einer Produktlinie im Vordergrund [4]. Es wird im FOSD-Prozess zwischen Problemraum und Lösungsraum unterschieden. Der Problemraum drückt sich u.a. in der Domänenanalyse aus, in der die Anforderungen für die gesamte Software-Produktlinie und die Produkte, die in der Produktlinie enthalten sein sollen, ermittelt werden. Das Ergebnis wird üblicherweise in einem Feature-Modell dokumentiert, aus dem Features und Constraints hervorgehen.

2.2.1 Feature-Modell

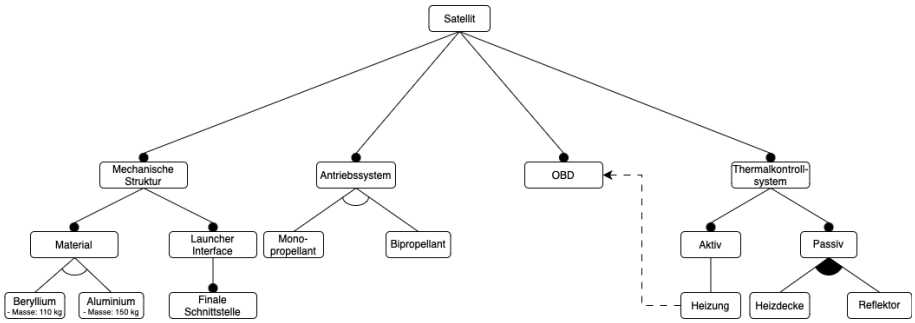


Abbildung 3: Beispiel eines Feature-Modells

Das Feature Modell ist ein Informationsmodell und repräsentiert alle logisch möglichen Produkte einer Software-Produktlinie. Es besteht aus Features und deren Beziehungen. Das Feature-Set ist hierarchisch angeordnet, sodass zum einen die Beziehungen zwischen Eltern-Feature und Kind-Feature beschrieben werden, aber zum anderen auch die Beziehungen zwischen einzelnen Teilbäumen. Das Wurzel-Feature ist in allen Produkten der Produktlinie enthalten. Das grundlegende Feature Modell umfasst vier Feature-Beziehungen.

- **Mandatory:** Das Kind-Feature muss zwangsweise ausgewählt werden, sobald das Eltern-Feature in das Produkt inkludiert wurde.
- **Optional:** Das Kind-Feature kann wahlweise ausgewählt werden, sobald das Eltern-Feature in das Produkt inkludiert wurde.
- **Alternative:** Aus einem Set an Kind-Features, die einem Eltern-Feature entstammen, kann genau ein Kind-Feature ausgewählt werden, sobald das Eltern-Feature inkludiert wurde.
- **Or:** Aus einem Set an Kind-Features können ein oder mehrere Kind-Features ausgewählt werden, sobald das Eltern-Feature inkludiert wurde.

Cross-Tree-Constraints beschreiben die Beziehungen zwischen Features, die in verschiedenen Teilbäumen auftreten. Diese Beziehung kann gerichtet sein, sodass das jeweilige Argument, nicht umgekehrt anwendbar ist. Hier ist zu unterscheiden zwischen:

- **Requires:** Dieses Inklusionsargument impliziert die Inklusion von Feature B, sobald Feature A ausgewählt wurde.

- **Excludes:** Dieses Exklusionsargument impliziert die Exklusion von Feature B, sobald Feature A ausgewählt wurde.

Erweiterte Feature-Modelle erlauben es, einem Feature Attribute zuzuweisen, mit welchen dann weitere Constraints aufgestellt werden können. Hat ein Feature z.B. ein Attribut Kosten, kann für einen Teilbaum des Modells ein Kostenbudget aufgestellt werden, welches dann Varianten, deren Features summiert dieses Budget überschreiten, invalidiert. Ein Feature-Modell beschreibt Konfigurationen, die aus einem Feature-Set bestehen. Benavides et al. [5] erklären Konfigurationen als ein 2-Tupel *(S,R)*, wobei S das Set aus selektierten und R das Set aus entfernten Features ist. Dabei gilt die Formel 1,

$$S, R \subseteq F. \tag{1}$$

wonach das Tupel der selektierten Features und der entfernten Features eine Teilmenge aller Features F ist. Weiterhin wird für die Konfigurationen zwischen Teil- und Vollkonfigurationen unterschieden. In einer Vollkonfiguration entspricht die Vereinigung aus selektierten und entfernten Features der Menge aller Features (2).

$$S \cup R = F. \tag{2}$$

In einer Teilkonfiguration entspricht die Vereinigung aus selektierten und entfernten Features einer echten Teilmenge aller Features (3), sie enthält also nicht alle Features.

$$S \cup R \subset F. \tag{3}$$

Ein Produkt ist äquivalent zu einer Vollkonfiguration, aus der das Set R entfernt wurde und nur die Features aus dem Set S definiert werden. Kann ein Feature-Modell kein valides Produkt hervorbringen, wird es als leeres (engl. void) Feature-Modell bezeichnet [5]. Ein Feature-Modell wird zumeist dann leer, wenn Cross-Tree-Constraints alle Feature-Kombinationen unmöglich werden lassen.
Die Beziehungen eines Feature-Modells können in die konjunktive Normalform übertragen werden, um das Modell hinsichtlich Validität und möglichen Varianten mit einem SAT-Solver auswerten zu können. In Tabelle 1 sind die entsprechenden Terme ([17]) aufgetragen.

Beziehung	Konjunktive Normalform
f_1 *Mandatory*-Feature von f	$(f_1 \vee \neg f) \wedge (\neg f_1 \vee f)$
f_1 *Optional*-Feature von f	$(\neg f_1 \vee f)$
$f_1, ..., f_n$ OR-Kind-Features von f	$(f_1 \vee \cdots \vee f_n \vee \neg f) \wedge \bigwedge_{i=1...n}(\neg f_i \vee f)$
$f_1, ..., f_n$ XOR-Kind-Features von f	$(f_1 \vee \cdots \vee f_n \vee \neg f) \wedge \bigwedge_{i=1...n}(\neg f_i \vee f) \wedge \bigwedge_{i<j}(\neg f_i \vee \neg f_j)$
f_1 Requires f_2	$(\neg f_1 \vee f_2)$
f_1 Excludes f_2	$(\neg f_1 \vee \neg f_2)$

Tabelle 1: Beziehungstypen im Feature-Modell und ihre Übersetzung in die konjunktive Normalform

3 Konzept

Der Rahmen des Konzepts wird zunächst in einer Software Requirements Specification gesetzt, worauf anschließend die Definitionen der User Stories und der konzeptionellen Anforderungen folgen. Die Grundlage für das Konzept wird in Kapitel 3.2 anschaulich am Beispiel eines kleinen Satellitensystems gezeigt. Danach wird der Kernpunkt des Konzepts vorgestellt, wie ein Feature-Modell als Ad-hoc Produktlinie fungieren kann. Die Einbindung des Konzepts in die Product Structure aus Virtual Satellite wird zuletzt aufgezeigt.

3.1 Software Requirements Specification nach IEEE

Die hier definierte SRS folgt den Kernpunkten einer SRS nach dem IEEE Guide aus [6]. Die prototypische Implementierung umfasst ein Plugin für Virtual Satellite, welches die Funktionen der Software um ein Feature-Modell erweitert, mit dem der Entwurfsprozess eines Satelliten abgebildet werden kann. Dieses Plugin basiert auf dem theoretischen Entwurf einer Ad-hoc-Produktlinie, die die während eines Entscheidungspunkts entstehenden Varianten des Satellitenentwurfs als Produkte einer Produktlinie mittels eines Variationspunkts abbildet. Diese Produktlinie bietet dem Ingenieur die Möglichkeit, Entscheidungen zeitlich zu verschieben und das automatisierte Filtern invalider Varianten. Der Ingenieur kann jederzeit eine Variantenselektion durchführen und sich für einen Entwurf in der Produktlinie entscheiden. Das Plugin kann, wie die anderen Virtual Satellite *Concepts*, über den *Active Concept Editor* aktiviert werden.
Die Produktfunktionen beinhalten eine Integration des Konzepts in den vorhandenen und von den Ingenieuren gewohnten Virtual Satellite-Workflow, ein dynamisches Feature-Modell als Ad-hoc-Produktlinie des Entwurfsprozesses und Features als Abbildung der Raumfahrzeugkomponenten, welche Attribute besitzen und Beziehungen eingehen können. Weiterhin können externe Beschränkungen zur Invalidierung von Varianten definiert werden. Automatisierte Funktionen umfassen zum einen die Veränderung der Ad-hoc-Produktlinie bei Änderung des Entwurfs durch den Ingenieur und eine Analyse des Feature-Modells auf valide Varianten und Darstellung aller während der Analyse gefundenen Varianten. Eine oder mehrere dieser validen Varianten können dann ausgewählt und als Configuration Tree instanziiert werden.
Die Zielgruppe sind Ingenieure mit Kenntnis im Satellitenentwurf. Wissen über Feature-Modelle muss nicht vorhanden sein. Zur erweiterten Nutzung des Plugins, d.h. explizite Instanziierung von Beziehungen, sollte der Nutzer die verschiedenen Beziehungscharakteristiken (XOR, OR, REQUIRES, EXCLUDES) verstehen können.
Da es sich um eine zeitlich eingeschränkte prototypische Implementierung handelt, ist die graphische Oberfläche rudimentär und rein funktionell zu verstehen. Außerdem wird auf die Implementierung von Kardinalitäten, Staged Configuration und automatische Benachrichtigungen für den Nutzer (z.B. "die aktuelle Änderung würde keine validen Varianten mehr zulassen") verzichtet. Externe Beschränkungen werden zum Funktionalitätsbeweis als Massenbudget implementiert.
Folgend werden die User Stories und spezifischen Anforderungen an das Konzept aufgelistet.

3.1.1 User Stories

N1 Als Ingenieur mit Fokus auf den korrekten Entwurf eines Raumfahrzeugs möchte ich meinen gewohnten Arbeitsablauf in Virtual Satellite gerne beibehalten.

N2 Als Ingenieur möchte ich einem Element physikalische Größen als Eigenschaft zuordnen können.

N3 Als Ingenieur möchte ich Beschränkungen wie Kostenfunktionen definieren und den Entwurf damit beeinflussen können.

N4 Als Ingenieur habe ich viel zu beachten, deswegen möchte ich, dass mir die Software sagt, wenn eine von mir getätigte Veränderung ein ungültiges Produkt entstehen lassen würde.

N5 Als Ingenieur im Entwurfsprozess muss ich viele Entscheidungen treffen, deswegen möchte ich alle aktuell möglichen Varianten meines Entwurfs einsehen können.

N6 Als Ingenieur benötige ich die volle Kontrolle über den Entwurfsprozess, deswegen möchte ich bei Bedarf Beziehungen und Hierarchien direkt definieren und verändern können.

N7 Wenn ich ein Element lösche, soll das Produkt mit dem gelöschten Element als Variante der Produktlinie gespeichert werden, sodass ich später darauf zugreifen kann.

N8 Wenn ich ein neues Element hinzufüge, soll das Produkt mit dem neuen Element als Variante der Produktlinie gespeichert werden, sodass ich später darauf zugreifen kann.

N9 Wenn ein Element die Präsenz eines anderen Elements erfordert, möchte ich diese Beziehung abbilden und die Elemente gemeinsam behandeln können.

N10 Wenn ein Element die Abwesenheit eines anderen Elements erfordert, möchte ich diese Beziehung abbilden und im Entwurfsprozess beachten können.

N11 Wenn ich Elemente im Configuration Tree anordne, sollen Elemente mehrmals vorkommen können.

N12 Wenn ich ein Produkt im Assembly Tree instanziieren möchte, will ich einfach eine Variante auswählen können.

3.1.2 Konzeptionelle Anforderungen

K1 Features müssen ein physisch oder logisch abgrenzbares Element eines Raumfahrzeuges repräsentieren können.

K2 Features müssen Beziehungen zu anderen Features eingehen können.

- K2.1 Hierarchische Beziehungen (Eltern-Kind) müssen abbildbar sein.
- K2.2 Grundlegende Beziehungen (Oder, Und, XOR) müssen abbildbar sein.
- K2.3 Baum-übergreifende Beziehungen müssen abbildbar sein.

K3 Features können Attribute beinhalten.

K4 Das Feature-Modell muss die Menge aller möglichen Varianten des Satellitenentwurfsprozesses abbilden.

K5 Das Feature-Modell soll den Entwurfsprozess nachbilden, deswegen soll es dynamisch verändert werden können.

K5.1 Features sollen dynamisch hinzugefügt werden können.

K5.2 Hierarchische Beziehungen sollen dynamisch verändert werden können.

K5.3 Beziehungen zwischen Sub-Features sollen dynamisch verändert werden können.

K5.4 Entfernte Features sollen mit veränderter Beziehung erhalten bleiben.

K6 Externe Beschränkungen müssen definiert werden und den Variantenraum beeinflussen können.

3.2 Szenarien für Entscheidungspunkte

In diesem Kapitel wird anhand eines kleinen beispielhaften Satellitenentwurfs ein Feature-Modell aus Entscheidungspunkten während des Entwurfsprozesses aufgebaut. Es soll deutlich gemacht werden, wie ein Feature-Modell dynamisch angepasst werden kann, um als Ad-hoc Produktlinie den natürlichen Entwurfsprozess nachzuempfinden.

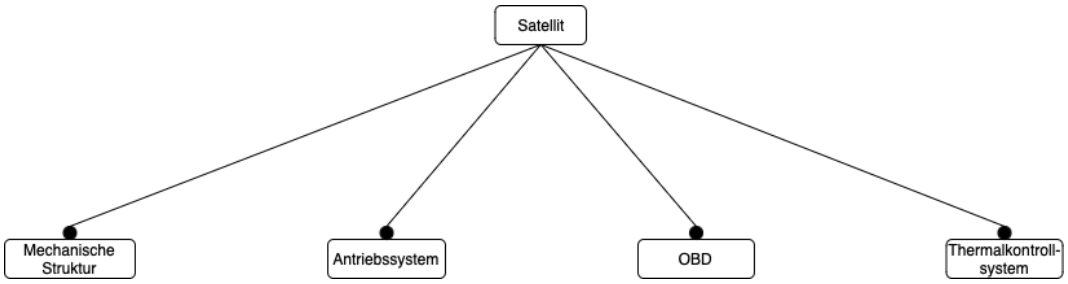


Abbildung 4: *Mandatory* Subsysteme

1. Der Ingenieur erstellt ein neues Projekt in Virtual Satellite und beginnt damit, Subsysteme zu dem Satelliten hinzuzufügen. Er erstellt einen Product Tree, den er ”Satellit” nennt und erstellt darin die Product Tree Domains ”Mechanische Struktur”, ”Antriebssystem”, ”OBD” und ”Thermalkontrollsystem”. Im Feature-Modell (Abb. 4) bildet das abstrakte Feature ”Satellit” den Wurzelknoten. Da die Subsysteme obligatorisch sind, werden sie als Kind-Features von ”Satellit” mit einer *Mandatory*-Beziehung abgebildet.

2. Nun geht der Ingenieur dazu über, das Material für die mechanische Struktur auszuwählen. Er fügt zur Product Tree Domain eine Eigenschaft für das Material hinzu und das Gewicht des jeweiligen Materials. Es kommt sowohl Beryllium mit einer Masse von 110 Kilogramm und Aluminium mit einer Masse von 150 Kilogramm in Frage. Im Feature-Modell (Abb. 5) entsteht ein neues Feature mit einer *Mandatory*-Beziehung zum Eltern-Feature ”Mechanische Struktur”. Dieses neue Feature hat als Kind-Features ”Beryllium” und ”Aluminium”, die mit einer *XOR*-Beziehung verknüpft sind und ihre jeweilige Masse als Attribut tragen. Weiterhin soll für die mechanische

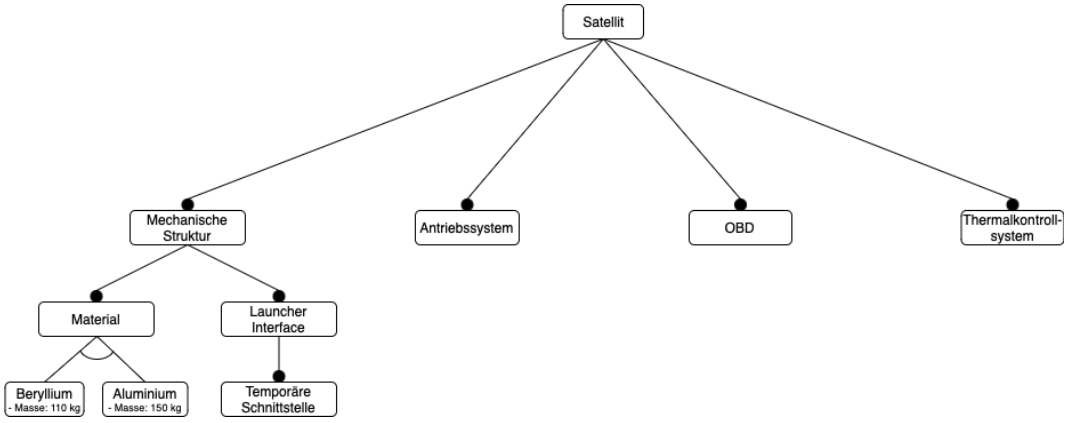


Abbildung 5: Mechanische Struktur

Struktur ein Launcher Interface implementiert werden, welches als Element Definition in der Product Tree Domain "Mechanische Struktur" erzeugt wird. Es ist allerdings noch nicht klar, welcher Launcher verwendet wird, deswegen wird die Element Definition "Temporäre Schnittstelle" erzeugt. Diese temporäre Schnittstelle wird als Kind-Feature an das Feature "Launcher Interface" angehängt.

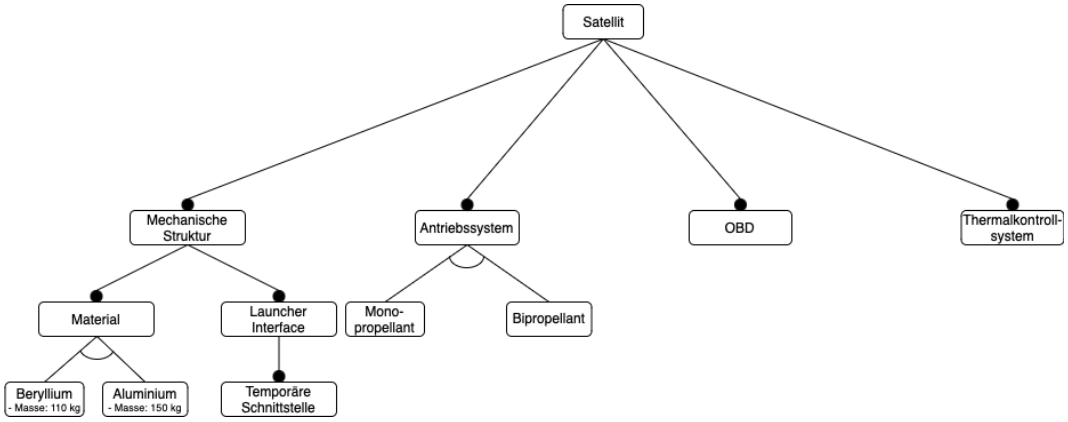


Abbildung 6: Antriebssystem

3. Da die Gesamtmasse des Satelliten noch nicht feststeht wird ein Monopropellant für das Antriebssystem ausgewählt und als Element Definition in der Product Tree Domain Antriebssystem erzeugt. Später stellt sich heraus, dass die Masse wahrscheinlich größer als gedacht ist, damit ein Monopropellant die Umlaufbahn effektiv beeinflussen kann. Als Alternative wird ein Bipropellant ausgewählt und zum Antriebssystem hinzugeügt. Die *XOR*-verknüpften Features "Monopropellant" und "Bipropellant" werden als Kind-Features an "Antriebssystem" angehängt (Abb. 6).

4. Für die passive Thermalkontrolle wird eine Thermaldecke verwendet. Die aktive Kontrolle wird mittels Heizungen durchgeführt, die über das OBD gesteuert werden. Als Alternative zur Thermaldecke werden Reflektoren zum Subsystem Thermalkontrolle hinzugefügt. Es ist nicht ganz klar, ob ein Element zur Kühlung ausreicht oder ob beide verwendet werden müssen. In der Product Tree Domain Thermalkontrollsystem werden zwei Product Tree Domains "Aktiv" und "Passiv" erstellt, wobei ersteres die

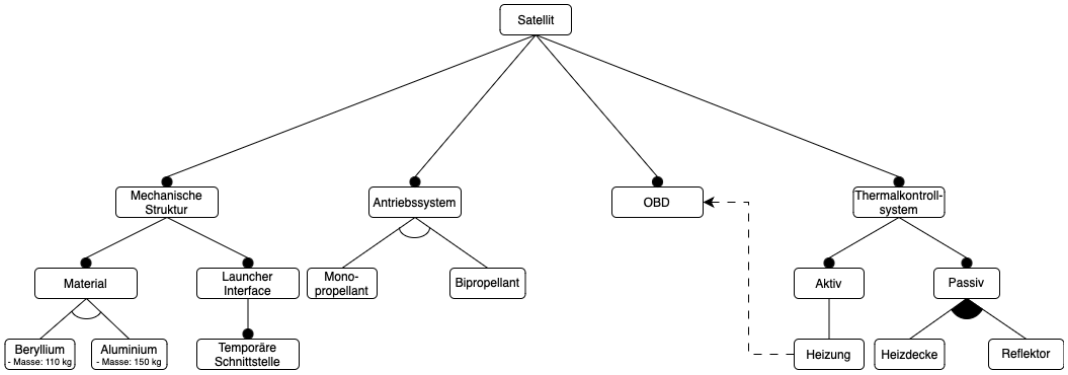


Abbildung 7: Thermalkontrolle

Element Definition ”Heizung” und letzteres die Element Definitions ”Heizdecke” und ”Reflektor” erhält. Im Feature-Modell (Abb. 7) werden dem Feature ”Thermalkontrollsystem” die obligatorischen Kind-Features ”Aktiv” und ”Passiv” hinzugefügt. Das Feature ”Aktiv” erhält ein Kind Feature ”Heizung”. Da die Heizung vom OBD gesteuert wird und damit die Existenz eines OBDs erfordert, wird eine *Requires*-Beziehung zwischen von dem Feature ”Heizung” zum Feature OBD erzeugt. Das Feature ”Passiv” erhält die Kind-Features ”Heizdecke” und ”Reflektor”, die mit einer *OR*-Beziehung verknüpft sind.

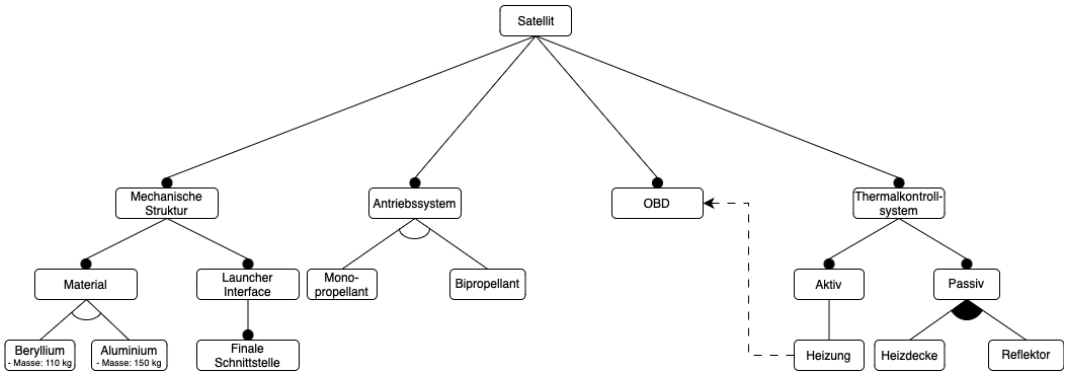


Abbildung 8: Austausch der Launcher-Schnittsteller

5. Der Launcher wurde festgesetzt, das temporäre Interface zum Launcher wird jetzt durch das richtige Interface ersetzt. In Virtual Satellite wird die Element Definition ”Temporäre Schnittstelle” gelöscht und durch ”Finale Schnittstelle” ersetzt. Im Feature Modell (Abb. 8) wird das Feature äquivalent ausgetauscht. -> Endgültiges Löschen des temporären Features

6. Die Gesamtmasse des Satelliten wurde jetzt definiert. Aufgrund des hohen Gewichts der anderen Subsysteme wird ein Limit von 120 Kilogramm für die mechanische Struktur festgelegt. Die Variante mit Aluminium ist nicht mehr zulässig. In Virtual Satellite wird in der Product Tree Domain ”Mechanische Struktur” ein Massenbudget von 120 Kilogramm erstellt. Dieses Budget wird in ein Constraint für das partielle Feature-Modell unterhalb des Features ”Mechanische Struktur” übersetzt und die Variante, bei der das Feature Aluminium ausgewählt werden würde, wird invalide (Abb. 9).

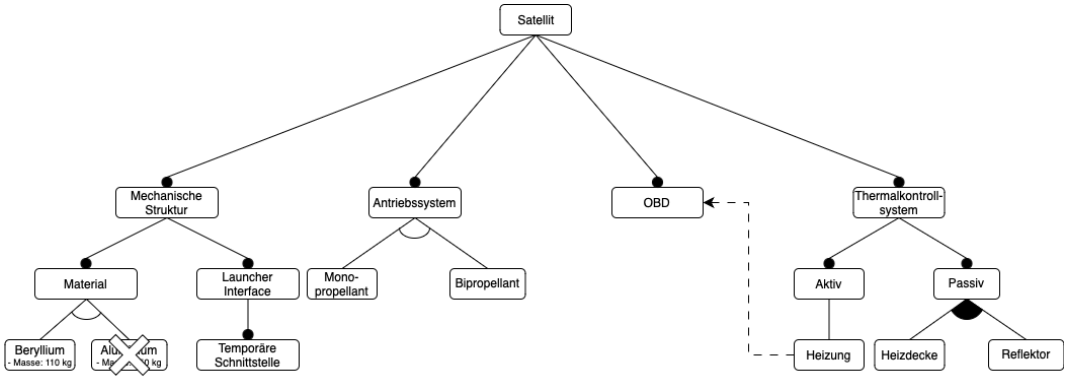


Abbildung 9: Aluminium wird invalide

3.3 Das Feature-Modell als Ad-hoc Produktlinie

Der aktuelle Entwurfsprozess in Virtual Satellite folgt einem iterativen Vorgehen. Sobald der Ingenieur eine Änderung an dem Entwurf vornimmt, verfällt der alte Entwurf. Tauscht er z.B. einen Antrieb mit flüssigem Treibstoff gegen einen mit festem Treibstoff, geht der Satellitenentwurf mit flüssigem Treibstoff verloren. Das wird vorwiegend zum Nachteil, wenn der Ingenieur sich aufgrund fehlender Informationen noch nicht sicher ist, welche Variante er bevorzugt. Eine Entscheidung zu einem späteren Zeitpunkt ist nur durch Austausch der Komponente, aller Subkomponenten und anschließender Überprüfung der Validität des Systems möglich. In Abb. 10 sind die Anzahl der während des Entwurfsprozesses verloren gegange-

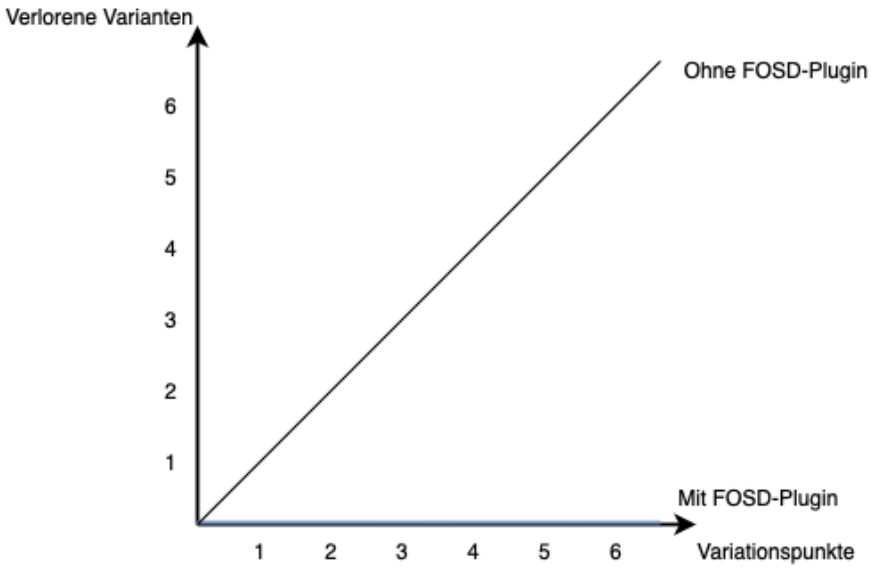


Abbildung 10: Verlorene Varianten im Entwurfsprozess

nen Varianten gegen den Fortschritt des Entwurfsprozesses aufgetragen. Der Fortschritt des Prozesses wird hier an Variationspunkten gemessen, also der Moment, an dem der Ingenieur sich zwischen verschiedenen Komponenten entscheiden muss. Die Abbildung vernachlässigt zur Anschaulichkeit Exponentialeffekte der Variantenbildung, die durch die Kombination mehrerer Beziehungen entstehen. Als simples Beispiel betrachte man das Feature-Modell

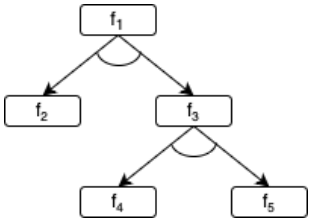


Abbildung 11: Simples Feature-Modell mit zwei Variationspunkten

in Abb. 11, das nur aus den Features $f_1..f_5$ besteht, wobei f_1 als Wurzel-Feature für die Bestimmung des Variantengrades unberücksichtigt bleibt. Dieses Feature-Modell zeigt zwei Variationspunkte in Form von XOR-Beziehungen zwischen f_2 und f_3, f_4 und f_5.

$$var(F) = \sum_{i=1}^{n} var(CF_i). \tag{4}$$

Die Bestimmung des Variantengrades folgt der Formel 4 ([12]), wobei *var(F)* der Variantengrad des Wurzel-Features bzw. damit des gesamten Feature-Modells und *var(CF)* der Variantengrad eines Kind-Features ist. Damit ergibt sich für dieses simple Feature-Modell mit zwei Variationspunkten bereits ein Variantengrad von 3, es existieren also drei Varianten. Weitere Beziehungen wie eine Optional-Beziehung bedienen sich des Multiplikationsoperators und erhöhen dementsprechend den Variantengrad um ein Vielfaches. In der Abbildung wird je Variationspunkt also nur eine neue Variante erzeugt. Der Gewinn an Varianten wird nichtsdestotrotz deutlich, da im iterativen VirSat-Entwurfsprozess ohne FOSD-Plugin der Variationsgrad stetig bei 1 verweilt; alte Varianten verfallen.
Um dem entgegenzuwirken, wird mit einem Feature-Modell der Entwurfsprozess abgebildet. Ein Entwurfsprozess ist dynamisch, Entscheidungen werden getroffen, Komponenten hinzugefügt, entfernt und verändert. Ein Feature-Modell ist hingegen eine Struktur, die in der FOSD nur unregelmäßig verändert wird. Der Entwurfsprozess kann also nur mit einem Feature-Modell dargestellt werden, sofern es dynamisch manipulierbar ist (Anforderung K6). Dem Feature-Modell können also jederzeit Features hinzugefügt werden, entnommen oder verändert werden so wie dem Satelliten im Product Tree EDs/PTDs hinzugefügt und entfernt werden. Die Veränderung des Product Trees ist dementsprechend mit einer Veränderung des Feature-Modells verknüpft, sodass die Varianten, die durch Veränderung des Product Trees entstehen und verloren gehen im Feature-Modell als Variante hinterlegt werden. Genauer bedeutet das, dass jede Operation im Product Tree auf eine Operation und damit auf eine Veränderung der Beziehung von Features im Feature-Modell abgebildet wird. In Kapitel 3.2 sind solche Operationen im Product Tree und die dazugehörige Veränderung im Feature-Modell dargestellt. Verallgemeinernd können wir folgende Operationen im Product Tree im Feature-Modell abbilden:

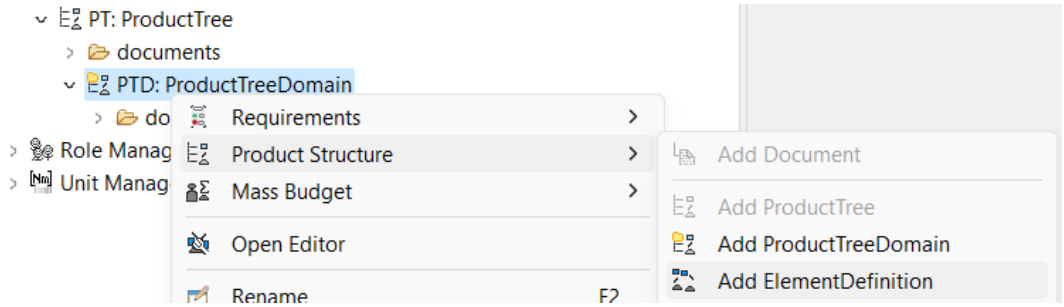


Abbildung 12: Hinzufügen einer Element Definition zu einer Product Tree Domain in Virtual Satellite

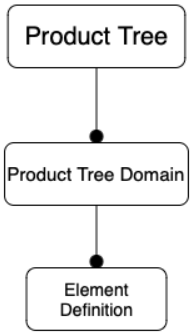


Abbildung 13: Darstellung der Operation als *Mandatory-Beziehung im Feature-Modell*

1. Wird dem Product Tree eine Product Tree Domain hinzugefügt, kann im Normalfall davon ausgegangen werden, dass diese Product Tree Domain zum aktuellen Zeitpunkt obligatorisch für alle Varianten des Entwurfs ist. Deswegen wird das Hinzufügen einer PTD auf eine *Mandatory*-Beziehung zwischen dem Eltern-Feature Product Tree und dem Kind-Feature Product Tree Domain abgebildet (Abb. 12 und 13). Analog kann beim Hinzufügen einer Element Definition zu einer Product Tree Domain davon ausgegangen werden, dass sie zum aktuellen Zeitpunkt für alle Varianten notwendig ist und auch diese Operation auf eine *Mandatory*-Beziehung zwischen dem Eltern-Feature Product Tree Domain und dem Kind-Feature Element Definition abgebildet wird. Für alle Varianten gilt also $A \subseteq S$ und folgend Gleichung 2, wobei A das Set aus hinzugefügten Features (PTD oder ED) und S, R die Sets der selektierten und entfernten Features ist. A ist definitiv in den selektierten Features enthalten.

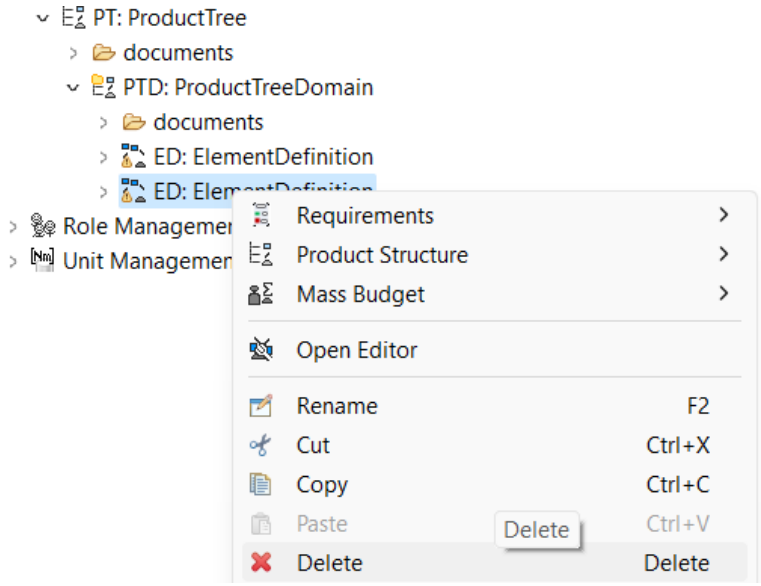


Abbildung 14: Entfernen einer Element Definition in einer Product Tree Domain in Virtual Satellite

2. Das Entfernen einer Product Tree Domain oder einer Element Definition (Abb. 14) kann mehrere Intentionen haben, die es alle zu implementieren gilt:

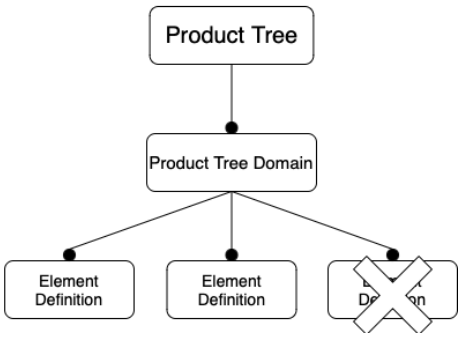


Abbildung 15: Effekt des Entfernens im Feature-Modell: Entfernen des Feature-Knotens

- Zum einen kann der Ingenieur beabsichtigen, den Knoten endgültig aus dem Product Tree zu löschen, sodass es in keiner Variante mehr vorkommt. Es gilt $\{PTD, ED\} \cap F = \emptyset$, wobei $\{PTD, ED\}$ das Set der Featurerepräsentationen der zugehörigen PTDs und EDs und F das Gesamtset aller Features ist. In diesem Fall wird das Entfernen der Knoten aus dem Product Tree auf das Entfernen der zugehörigen Features aus dem Feature-Modell abgebildet (Abb. 15).

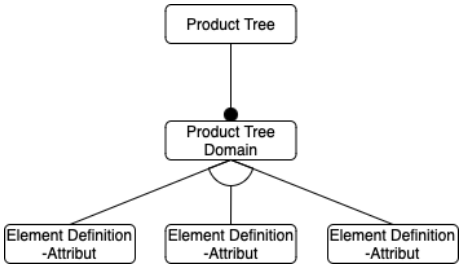


Abbildung 16: Effekt des Entfernens im Feature-Modell: *XOR*-Beziehung

- Die zweite Intention ist das eventuelle Ersetzen der PTD oder ED mit einer anderen, d.h. hier ist ein Entscheidungspunkt zwischen zwei oder mehr alternativen Knoten. Dieser Entscheidungspunkt wird im Feature-Modell als eine *XOR*-Beziehung zwischen dem Feature, das zu dem entfernten Knoten gehört, und den anderen Features der selben Ebene abgebildet (Abb. 16). Hier gilt $\{PTD, ED\} \subset S \Delta R$, wobei $\{PTD, ED\}$ das Set der Featurerepräsentationen der entfernten und nicht entfernten Features der selben Ebene, S die selektierten und R die entfernten Features darstellt. So können die Features $\{PTD, ED\}$ entweder in den selektierten Features oder in den entfernten Features auftauchen.

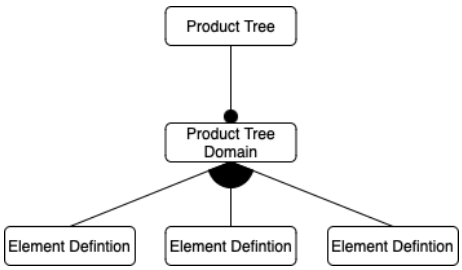


Abbildung 17: Effekt des Entfernens im Feature-Modell: *OR*-Beziehung

- Die dritte Intention ist das Ersetzen der PTD oder ED mit einer anderen oder das Kombinieren mit anderen PTDs und EDs. Das heißt, hier kann es zusätzlich zu den alternativen Knoten noch Kombinationen aus den Knoten einer Baumebene geben, sodass bei einer Baumebene aus zwei Kind-Features drei Varianten entstehen würden: Entweder Knoten eins oder Knoten zwei oder beide Knoten zusammen. Im Feature-Modell wird das mit einer *OR*-Beziehung abgebildet (Abb. 17). Es kommen in einer Variante also mindestens eines der Features und maximal n Features vor, mit n = Anzahl der Kind-Features in der Ebene. Es gilt $\{PTD, ED\} \subset S \cap R$, mit obigen Variablendeklarationen. Die PTD- und ED-Features können damit sowohl in den selektierten Features oder den entfernten Features als auch in beiden Sets vorkommen.

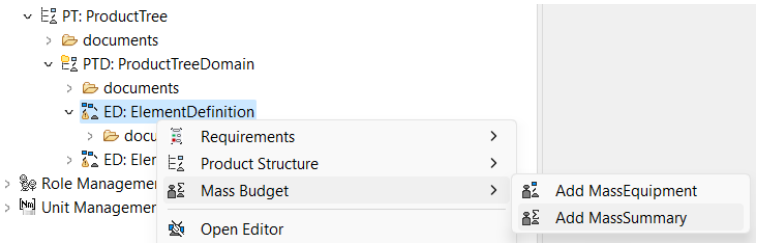


Abbildung 18: Hinzufügen eines Attributs zu einer Element Definition im Product Tree

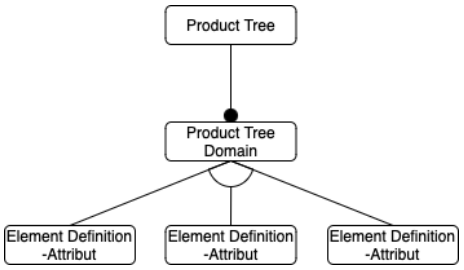


Abbildung 19: Attribute der Features im Feature-Modell

3. Einer PTD oder ED ein Attribut zuweisen führt ebenfalls dazu, dass der zugehörigen Featurerepräsentation ein Attribut zugewiesen werden kann, was dann zur Definition weiterer Constraints genutzt werden kann (Abb. 18 und 19).

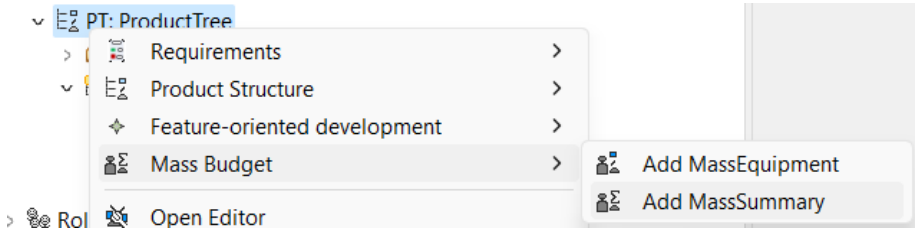


Abbildung 20: Hinzufügen eines Massenbudgets in Virtual Satellite

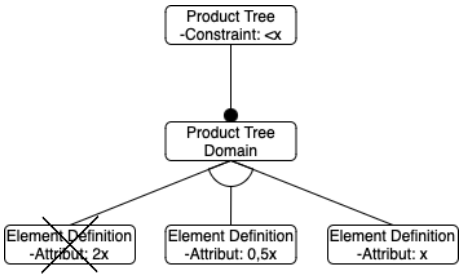


Abbildung 21: Der Constraint im Feature-Modell hat die Invalidität der Variante mit der linken Element Definition zur Folge

4. Einem Product Tree oder einer Product Tree Domain kann ein Budget wie z.B. eine Kostengrenze oder Massegrenze zugewiesen werden (Abb. 20). Im Feature-Modell wird dieses Budget als externes Constraint abgebildet. Das gesamte Modell oder ein partielles Modell für das dieses Constraint gesetzt wurde, wird auf die Verletzung dieser

Beschränkung überprüft und Varianten, die diese Verletzung begehen, entfernt (Abb. 21). Die Attribute aus Punkt 3 können zur Aufstellung und Überprüfung solch eines Constraints genutzt werden, indem sie z.B. einen Massewert für das jeweilige Feature deklarieren.

Weiterhin existieren Operationen, die nicht im Product Tree stattfinden, aber explizit im Feature-Modell definiert werden können, um direkt einen Entscheidungspunkt zu dokumentieren oder Beziehungen kenntlich zu machen: Ein Entscheidungspunkt muss nicht zwingend automatisiert aus dem Entwurfsprozess hervorgehen und den Prozess dokumentieren. Der Ingenieur kann sich ebenso bewusst dafür entscheiden, dass er sich an einem Entscheidungspunkt befindet und die Alternativen selbstständig erfassen. Dazu kann er dem Feature-Modell neue Features einfügen und Beziehungen zu den bereits vorhandenen Features instanziieren. Wenn er z.B. weiß, dass die Wahl aus zwei Features besteht, kann er diese beiden Features zum Feature-Modell hinzufügen und mit einer *XOR*-Beziehung den Entscheidungspunkt mit zwei Alternativen markieren. Darüber hinaus sind Cross-Tree-Constraints nicht im Product Tree vorgesehen und müssen dementsprechend direkt im Feature-Modell definiert werden.
Während des Entwurfsprozesses entstehen also natürlicherweise Beziehungen zwischen den Komponenten des Raumfahrzeuges. Diese Beziehungen können dazu genutzt werden, ein Feature-Modell aufzuspannen und alle Komponenten, die während des Entwurfsprozesses in Erwägung gezogen wurden, zu dokumentieren und den Zugriff darauf zu behalten. Dieses Feature-Modell bildet in seiner Gesamtheit alle validen Entwürfe des Prozesses ab und schafft damit eine kurzfristig manipulierbare ("Ad-hoc") Produktlinie.

3.4 FOSD-Einbindung in Virtual Satellite

Aus Anforderung N1 ergibt sich, dass die Aufgliederung in Product Tree, Configuration Tree und Assembly Tree weitestgehend erhalten bleiben soll. An erster Stelle des Entwurfsprozesses steht die Definition der Elemente des Raumfahrzeugs im Product Tree. Es werden neue Elemente erzeugt und mit Eigenschaften versehen. Dieser erste Schritt ist deckungsgleich mit dem Aufspannen eines Feature Modells. Die Features müssen zunächst mit Wissen aus der Domäne definiert werden. So bietet sich an, den Product Tree für die Definition der Features und ihrer Eigenschaften zu nutzen und dem Ingenieur so keinen weiteren Overhead aufzubürden. Weiterhin ordnen sich die Features durch die Baumtopologie des Product Trees auch direkt in eine Hierarchie ein, Eltern-Kind-Beziehungen entstehen.

3.5 Features

Wie im Kapitel 2.2 bereits aufgezeigt, herrscht eine Spannung bezüglich der Aufgaben von Features, die bei Verlagerung in den Problemraum die Definition abstrakter Konzepte in der Zieldomäne und bei Verlagerung in den Lösungsraum die Implementierung dieser Konzepte bedeutet. Als virtuelle Konzeption eines Raumfahrzeuges liegt der Schwerpunkt von Virtual Satellite im Problemraum, weswegen unser Fokus auf der Definition abstrakter Features (d.h. Repräsentationen physischer/software-technischer Bauteile) liegt, die Bestandteil eines Raumfahrzeugs sein können. Die tatsächliche Implementierung der Features überlassen wir den (Software-) Ingenieuren der Hersteller. Features zeigen, was von einem System erwartet wird, um das Ziel zu erreichen. In Virtual Satellite repräsentieren die Features also

die Erwartungen an das Raumfahrzeug, um die angedachte Mission zu erfüllen. Weiterhin muss beachtet werden, dass eben schon eine fertige und aktiv genutzte Software vorliegt. Die Nutzer sind an den Arbeitsablauf von Virtual Satellite gewöhnt, jede Änderung am Entwurfsprozess würde den Fokus vom Entwurf eines korrekten Raumfahrzeugs zum Erlernen des neuen Prozesses umschwenken und das Resultat eventuell negativ beeinflussen. Deswegen steht als eine der wichtigsten Anforderungen (Anforderung N1) der größtmögliche Beibehalt des Entwurfsprozesses im Vordergrund, wie er vor Beginn dieser Arbeit im VirSat Core vorgefunden wurde. Das FOSD-Plug-In arbeitet also mit den Nutzereingaben, die auch in der Core-Version getätigt würden. Es erfolgt demnach ein eins-zu-eins Mapping einer Element Definition zu einem Feature. Die Anforderungen K1, K2.1 und K3 werden bereits von Element Definitions erfüllt, da sie in einer Baumtopologie angeordnet sind und Attribute wie z.B. Raummaße oder Masse besitzen können und können damit als Informationsgrundlage dienen. So erbt durch das eins-zu-eins Mapping ein Feature seine hierarchische Beziehung und Attribute von einer Element Definition. Eine Element Definition repräsentiert entweder eine physische oder eine Software-Komponente und repräsentiert damit ein physisch oder logisch abgrenzbares Element eines Raumfahrzeuges. Das Feature erbt ebenfalls diese Repräsentation und ist damit selbst ein abgrenzbares Element.

3.6 Feature Tree

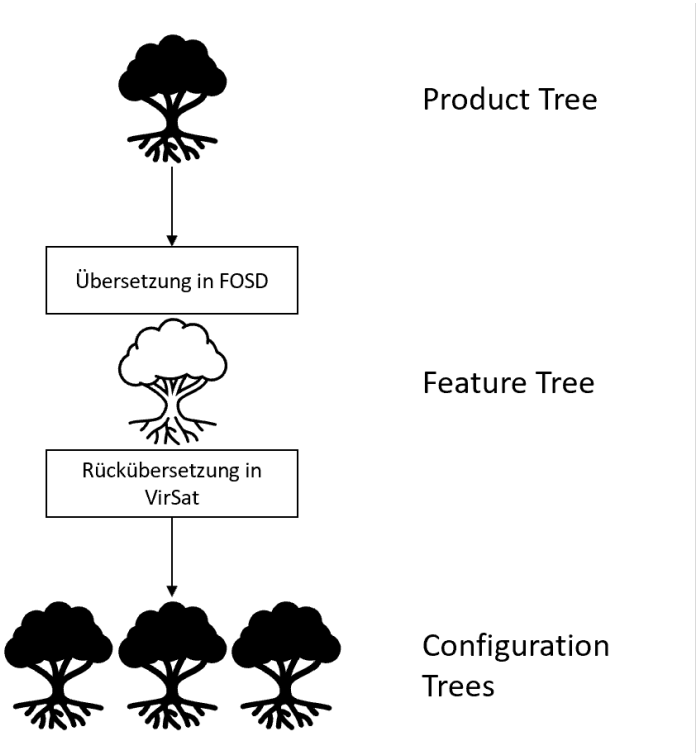


Abbildung 22: VirSat-Baumstruktur mit eingeschobenem Feature Tree

Virtual Satellite liefert eine Baumstruktur für den Entwurf von Raumfahrzeugen (s. Kapitel 2.1.1). Die Core-Version lässt genau einen Product Tree und einen Assembly Tree, aber mehrere Configuration Trees zu. Im Configuration Tree sind also mehrere mögliche

Entwürfe anzutreffen. In die Sprache der Feature-orientierten Entwicklung übersetzt, können an dieser Stelle mehrere Varianten einer Produktlinie erstellt werden. Diese Varianten bilden die Vorlage für die tatsächlichen Instanzen des Entwurfs, die wie gehabt im Assembly Tree instanziiert werden. Findet die Variantenselektion in der Phase des Configuration Trees statt, muss folglich das Feature Modell vorher aufgebaut werden, denn es spannt den gesamten Variantenraum auf. Nun erscheint der Sprung vom Product Tree, der lediglich die Features und Eltern-Kind-Beziehungen beinhaltet, zu einer komplettierten Produktlinie mit mehreren möglichen Varianten sehr groß. Es fehlt ein fließender Übergang, der alle Arten von Feature-zu-Feature-Beziehungen aufzeigen und das gesamte Feature-Modell aufspannen kann: Der Feature Tree. Er erbt vom Product Tree und übernimmt dementsprechend auch alle Änderungen, die dort (auch retrospektiv) geschehen. Er übersetzt die VirSat-Elemente in Elemente aus der FOSD und erstellt daraus ein Feature-Modell. Der Feature-Tree lässt sich explizit verändern, um dem Ingenieur volle Kontrolle über den Prozess zu geben. Bei der Implementierung ist darauf zu achten, dass Veränderungen im Feature Tree und retrospektive Veränderungen im Product Tree nicht in Konflikt geraten. Ein Configuration Tree erbt die hierarchischen Beziehungen vom Feature Tree und nimmt eine in VirSat-Elemente zurückübersetzte Variante des Feature-Modells an, die dann in aus der Core-Version gewohnter Weise dargestellt wird. Der Feature Tree hat also zwei Schnittstellen mit zwei unterschiedlichen Übersetzungsfunktionen. Die Schnittstelle zum Product Tree nimmt die VirSat-Elemente, übersetzt sie in die FOSD-Sprache und baut daraus ein Feature-Modell. Die Schnittstelle zu den Configuration Trees entnimmt dem Feature-Modell eine Variante, übersetzt die FOSD-Elemente in VirSat-Elemente und übergibt sie einem Configuration Tree. Abbildung 23 zeigt die beispielhafte Struktur eines Feature Trees. Er bietet folgende

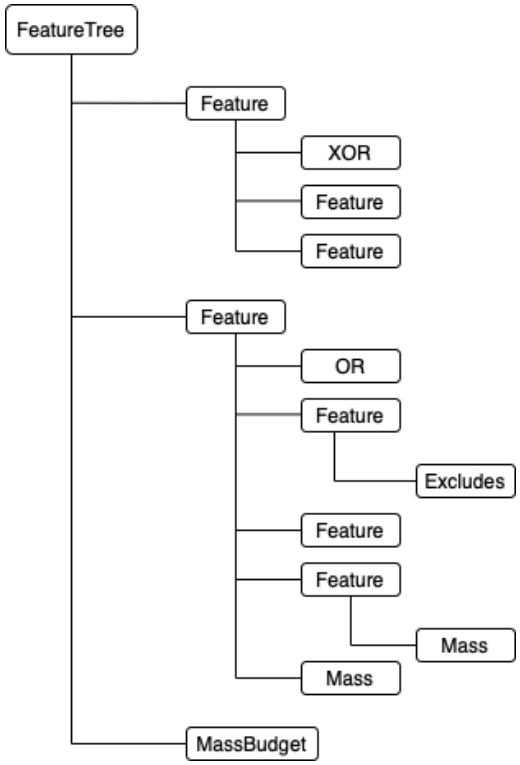


Abbildung 23: Struktur eines Feature Trees

Bestandteile:

- Der Wurzelknoten *FeatureTree* bildet den Container für alle weiteren Features und repräsentiert damit das Raumfahrzeug in seiner Gesamtheit (Anforderung K1).
- Diverse Feature-Knoten sind in dem FeatureTree angeordnet und repräsentieren Bestandteile des Raumfahrzeugs (Anforderung K1).
- Die Feature-Knoten gehen mit anderen Feature-Knoten und dem FeatureTree-Knoten hierarchische Beziehungen ein (Anforderung K2.1).
- Feature-Knoten auf einer Ebene gehen Beziehungen mit den anderen Feature-Knoten auf einer Ebene ein wie z.B. *XOR, OR* (Anforderung K2.2). Diese Beziehungen sind als eigene Knoten selbst Bestandteil der Baumstruktur.
- Cross-Tree-Constraints sind ebenfalls eigene Knoten wie hier als *Excludes* zu sehen (Anforderung K2.3).
- Feature-Knoten haben ihre Attribute untergeordnet als Attribut-Knoten wie hier beispielhaft als *Mass* dargestellt (Anforderung K3).
- Externe Beschränkungen gelten für das gesamte Raumfahrzeug und werden deshalb als Knoten unterhalb des FeatureTrees instanziiert wie hier am *MassBudget* zu sehen ist (Anforderung K6).

Diese Bestandteile des Feature Trees müssen nun in ein Feature-Modell übersetzt werden, um Varianten auswählen und daraus Configuration Trees erstellen zu können. Tabelle 2 stellt die Bestandteile eines Feature Trees den übersetzten Bestandteilen des Feature-Modells gegenüber.

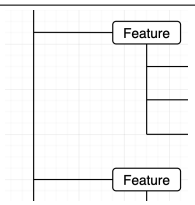
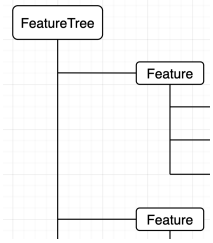
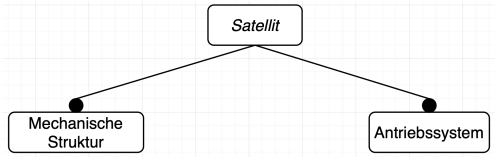
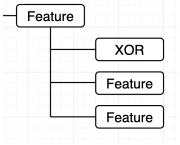
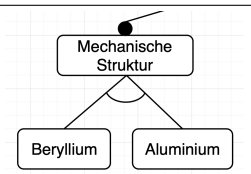
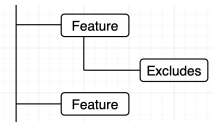
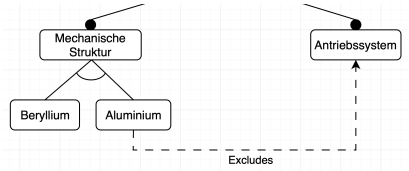
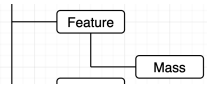
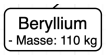
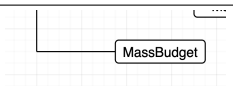
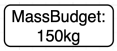
	Feature Tree	Feature-Modell
Wurzelknoten		
Features		
Hierarchie		
Beziehungen einer Ebene		
Cross-Tree-Constraints		
Attribute		
Externe Constraints		

Tabelle 2: Mapping Feature Tree zu Feature-Modell

3.6.1 Bauminteraktion

Der Feature Tree fügt sich nahtlos in den Arbeitsablauf von Virtual Satellite ein. Das Sequenzdiagramm in Abb. 24 zeigt den Ablauf der Interaktionen zwischen den Bäumen im Repository eines Projekts in Virtual Satellite. Zu Beginn existiert ein Repository, in dem ein Product Tree generiert und hinzugefügt wird (*generateProductTree*). Dem Product Tree werden dann diverse PTDs und EDs hinzugefügt. Ausgehend vom Product Tree kann dann ein Feature Tree generiert und dem Repository hinzugefügt werden (*generateFeatureTree*). Der Feature Tree erbt die Knoten, Attribute und hierarchischen Beziehungen vom Product Tree. Anschließend kann auch weiter am Product Tree gearbeitet, Knoten hinzugefügt, entfernt oder manipuliert werden. Um den Feature Tree wieder mit dem Product Tree zu synchronisieren, kann er aktualisiert werden und erhält vom Product Tree die Knoten, die hinzugefügt, entfernt oder manipuliert wurden und pflegt sie ein (*updateFeatureTree*). Anschließend wird der Feature Tree ausgewertet, seine Knoten und Beziehungen werden in ein

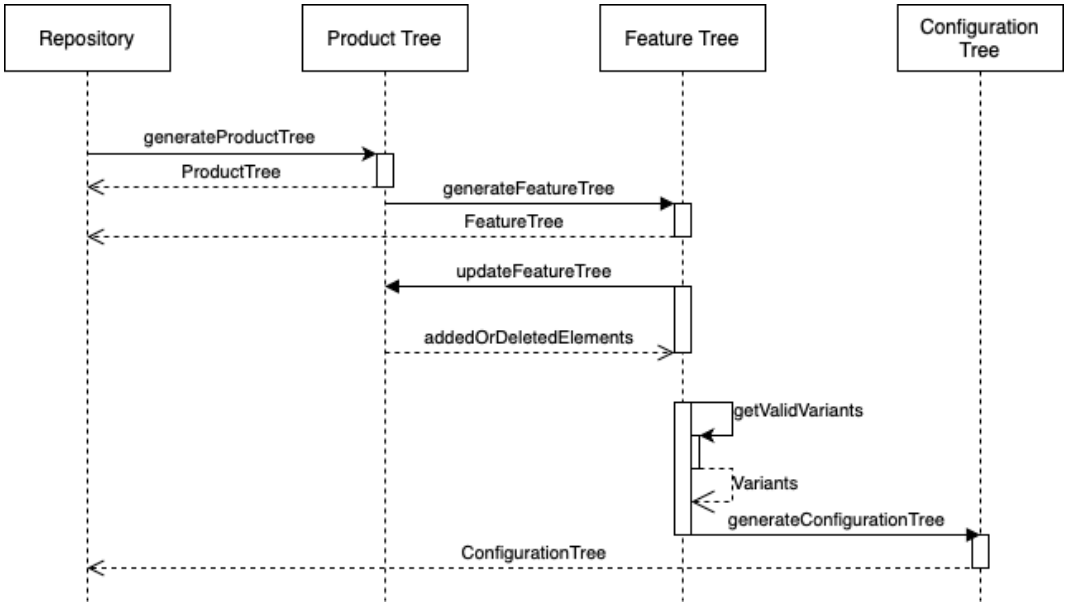


Abbildung 24: Sequenzdiagramm der Interaktion zwischen den Bäumen des Repositorys

Feature-Modell übersetzt und die validen Varianten ermittelt (*getValidVariants*). Aus den validen Varianten kann eine selektiert werden und als Configuration Tree instanziiert und dem Repository hinzugefügt werden (*generateConfigurationTree*).

4 Implementierung

Zunächst werden die technischen Anforderungen an das Plugin aufgelistet, worauf die Erklärung der Generic System Engineering Language folgt. Anschließend werden die Datei concept.concept, der Feature Tree und die Auswertung des Feature Trees anhand des Codes erklärt. Zuletzt wird aufgezeigt, wie die Varianten selektiert und als Configuration Tree erzeugt werden können.

4.1 Technische Anforderungen

1. Das Plug-In soll mit einem Concept realisiert werden.
2. Die Generic System Engineering Language wird als Grundlage verwendet.
3. Das Plug-In soll weitestgehend modular sein, d.h. bestehende Concepts sollen nicht verändert werden.
4. Der Feature Tree soll aus dem Product Tree generiert werden können.
5. Der Feature Tree soll bei Veränderung des Product Trees aktualisiert werden, weitere Features werden über neue Element Definitions im Product Tree hinzugefügt.
6. Das Hinzufügen und Entfernen einer neuen Element Definition nach Erzeugung des Feature Trees soll zu einer auswählbaren Veränderung des Beziehungs-Attributs im Elternknoten führen.
7. Ein oder mehrere Configuration Trees sollen aus dem Feature Tree generiert werden können.
8. Beim Generieren eines Configuration Trees sollen mögliche Varianten angezeigt und ausgewählt werden können.
9. Die Generierung soll ähnlich zum Concept ProductStructure mittels Wizard erfolgen.
10. Die Features sollen ihre Hierarchie aus den Element Definitions des Product Trees erben.
11. Die Features haben ein Attribut, das ihre Abhängigkeit zu einem anderen Feature speichert.
12. Der Elternknoten hat ein Attribut zum Definieren von Beziehungen der Kindknoten (Oder, Und, XOR)
13. Die Features haben Attribute für ihre physischen/logischen Eigenschaften

4.2 Generic Systems Engineering Language

Die Generic Systems Engineering Language (GSEL) bietet die semantische Grundlage für das Erstellen von Domänenelementen. Die Wurzel des Concepts bildet die o.g. concept.concept Datei. Sie kann die Meta-Informationen *name*, *displayname*, *description* und *version* haben. In dieser Datei werden *StructuralElements* (SE) und *CategoryAssignments* (CA) definiert. Mittels der SEs kann das System in seine einzelnene Bestandteile zerlegt werden. Die CAs

können den SEs zugewiesen werden und beinhalten Informationen über diese Elemente. Diese Unterteilung dient vor allem dazu, die Engineering-Domäne und das Conceptual Data Model zu trennen. Weiterhin bietet es den Vorteil, modular Concepts hinzufügen und entfernen zu können.

Listing 1: Eine einfache Dekomposition

```
1  StructuralElement ComponentType {
2          IsRootStructuralElement;
3          Applicable For [ComponentType];
4  }
5
6  StructuralElement Component {
7          IsRootStructuralElement;
8          Inherits From [ComponentType];
9          Applicable For [Component];
10 }
```

StructuralElements können voneinander erben, um Informationen weiterzugeben. Ein SE hat ebenfalls die Attribute *name*, *shortname* und *description*. Mit dem Attribut *isRootStructuralElement* kann von dem SE direkt eine Instanz innerhalb des Repositorys erstellt werden, d.h. es kann als Wurzelknoten innerhalb des Projekts verwendet werden. Das SE ProductTree hat z.B. *isRootStructuralElement* als Attribut, wodurch eine Instanz des ProductTrees direkt im Repository erzeugt werden kann. Mit dem *Attribut Applicable For* wird definiert, welches SE diesem SE übergeordnet, d.h. ein Eltern-SE sein darf. Das SE ProductTreeDomain kann z.B. nur einen ProductTree als Eltern-SE haben, eine ElementDefinition hingegen sowohl eine ProductTreeDomain als auch eine andere ElementDefinition. In Listing 1 wird eine simple Dekomposition eines Systems dargestellt. Die StructuralElements *ComponentType* und *Component* sind definiert. Mit den in Zeile 2 und 7 definierten Attributen *isRootStructuralElement* können Instanzen der beiden SEs als Wurzelknoten im Repository fungieren. Mit dem Attribut *Applicable For* wird festgesetzt, dass ein *ComponentType* innerhalb eines anderen *ComponentTypes* und ein *Component* innerhalb eines anderen *Components* hinzugefügt werden kann. Mit *Inherits From [ComponentType]* in Zeile 8 wird außerdem noch definiert, dass *Component* seine Informationen vom *ComponentType* erben kann.

Kategorien ermöglichen das Hinzufügen von Domäneninformationen zu einem SE und werden mit einem CategoryAssignment instanziiert. Neben *Properties*, die die eigentlichen Informationen enthalten, kann ein CA die bereits erwähnten Attribute *name*, *shortname*, *description* und *Applicable For* haben. Mit Attribut *extends* kann eine Kategorie eine andere Kategorie erweitern und die Properties erben. Relevant für die Implementierung ist außerdem noch *Cardinality*, mit der die maximale Anzahl dieser Kategorie für ein SE definiert werden kann. Ohne Kardinalität existiert keine Begrenzung.

Wie bereits erwähnt, beinhalten Properties die Domäneninformationen und können als *Property Instance* instanziiert werden. Zahlen können als *FloatProperty* oder *IntegerProperty* definiert werden. Wichtige Attribute ist zum einen *default*, mit dem ein Standardwert nach der Instanziierung festgelegt werden kann und *unit*, um die Einheit des Werts anzugeben.

Weiterhin sind *BooleanProperty* und *StringProperty* und *EnumProperty* (mit selbst definierten Werten) möglich.

4.3 Plug-In

Die Entwicklung eines Eclipse-Plugins erfolgt üblicherweise mittels der Eclipse Rich Client Platform (Eclipse RCP). Darauf basierend hat das DLR eine eigene RCP entwickelt, um die bestehende Virtual Satellite Software um weitere Plugins - sogenannte *Concepts* - zu erweitern. Virtual Satellite-DEV-IDE bietet die Möglichkeit, schnell das Grundgesrüst eines Concepts zu bauen. Dafür erstellt es die nötige Ordnerstruktur und Klassen bzw. Dateien. Das Plugin ist in vier Packages unterteilt. Die Implementierung des FOSD-Plugins nutzt die vorhandene Struktur des Product-Structure-Plugins [1], um vom Product Tree, den Product Tree Domains und den Element Definitions zu erben und um einen Configuration Tree und Element Configurations vom Feature Tree und den Features erben zu lassen. Deswegen muss das Product-Structure-Package in den Klassenpfad der FOSD [2]- und FOSD-UI [3]- Packages aufgenommen werden. Als externe JAR-Library wird der Sat4j-Core (http://www.sat4j.org/products.php#core) zum Auswerten der Feature-Modelle verwendet.

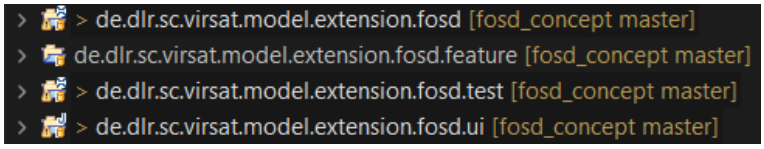


Abbildung 25: Package-Struktur des FOSD-Plugins

4.3.1 de.dlr.sc.virsat.model.extension.fosd

Dieser Ordner enthält u.a. den weiteren Ordner concept und den src bzw. src-gen Ordner. In der concept.concept Datei des concept Ordnes werden die Domänenelemente mittels der Generic Systems Engineering Language erstellt. Die Domänenelemente werden dann automatisch als Beans im src/model und src-gen/model (abstrakte Klassen) Ordner generiert. Diese Beans sind Java-Klassen, die für die Logik des Plugins verwendet werden können. Im folgenden ist die concept.concept Datei in ihre Elemente unterteilt dargestellt.

Listing 2: Structural Element *FeatureTree*

```
1 // --- Feature Tree ---
2 StructuralElement FeatureTree description "Feature tree to
3 describe common and variant spacecraft components."
4 {
5         Inherits From [ProductTree];
6         IsRootStructuralElement;
7 }
```

An erster Stelle ist der *FeatureTree* als Structural Element definiert. *Inherits From [ProductTree]* lässt ihn seine Meta-Informationen, Kind-Elemente sowie Category Assignments

[1] de.dlr.sc.virsat.model.extension.ps
[2] de.dlr.sc.virsat.model.extension.fosd
[3] de.dlr.sc.virsat.model.extension.fosd.ui

vom Product Tree erben. Mittels *isRootStructuralElement* wird der Feature Tree zur Wurzel der Baumstruktur, d.h. er kann kein Eltern-Element haben.

Listing 3: Structural Element *Feature*

```
1 // --- Feature ---
2 StructuralElement Feature description "Represents a spacecraft
3 componenent."
4 {
5         Inherits From [ElementDefinition, ProductTreeDomain];
6         Applicable For [FeatureTree, Feature];
7 }
```

Das Structural Element *Feature* kann von einer ElementDefinition oder einer ProductTree-Domain seine Meta-Informationen usw. erben und kann mit den Schlüsselworten *Applicable For [FeatureTree, Feature]* lediglich ein Kind-Element eines Feature Trees und Features sein.

Listing 4: Category *SubFeatureRelationship*

```
1 // --- SubFeatureRelationship ---
2 Category SubFeatureRelationship description "Category to describe
3 the relationship of the subfeatures."
4 {
5         Applicable For [Feature, FeatureTree];
6         Cardinality 1;
7         EnumProperty character values [or = "OR", xor = "XOR"];
8 }
```

Anschließend folgen die Categorys, mit denen wir dem Feature Tree und dem Feature Informationen zuweisen können. Die *SubFeatureRelationship* kann jeweils Attribut eines Feature Trees oder Features sein, dabei durch *Cardinality 1* aber lediglich genau einmal in einem Structural Element vorkommen. Mit der *EnumProperty character* kann sie die Beziehungen OR und XOR ausdrücken.

Listing 5: Category *OptionalRelationship*

```
1 Category OptionalRelationship description "Use this to mark
2 the feature as optional."
3 {
4         Applicable For [Feature];
5         Cardinality 1;
6         BooleanProperty isOptional default true;
7 }
```

Die Category *OptionalRelationship* kann nur dem Feature zugewiesen werden, da ein Feature Tree als Wurzelknoten nicht optional sein kann. Die Optionalität kann nur einmal je Feature vorkommen und hat die *BooleanProperty isOptional*, deren Standardwert true ist.

Listing 6: Category *CrossTreeConstraint*

```
1 Category CrossTreeConstraint description "Reference to another
2 feature and character of relationship."
```

```
3 {
4         Applicable For [Feature];
5         StringProperty referenceUuid;
6         EnumProperty character values [enumValue1 = "REQUIRES",
7         enumValue2 = "EXCLUDES"];
8 }
```

Die Category *CrossTreeConstraint* kann ebenfalls nur dem Feature zugewiesen werden, da eine excludes-Beziehung zwischen dem Wurzelknoten eines Feature-Modells und einem Feature des selbigen Modells sinnlos wäre und eine requires-Beziehung als mandatory-Beziehung ausgedrückt wird. Ein *CrossTreeConstraint* hat eine *StringProperty referenceUuid*, die die UUID des Features beinhaltet, mit dem das Feature mit dem *CrossTreeConstraint* eine Beziehung eingeht. Weiterhin hat diese Category eine *EnumProperty character*, um die beiden Beziehungstypen REQUIRES und EXCLUDES abzubilden.

Listing 7: Category *MassBudget*

```
1 Category MassBudget description "Max cumulated mass"
2 {
3         Applicable For [FeatureTree];
4         Cardinality 1;
5         FloatProperty mass;
6 }
```

Die Category *MassBudget* kann einem Feature Tree zugewiesen werden, um allen Elementen des Feature Trees eine kumuliertes Gesamtgewicht zu geben. Es kann nur einmal je Feature Tree vorkommen und hat eine *FloatProperty mass*, in der die Gesamtmasse eingetragen wird.

4.3.2 de.dlr.sc.virsat.model.extension.fosd.feature

Dieses Package verpackt das .fosd Package und das .ui Package in ein gemeinsames Plugin, das über den Active Concepts Editor in VirSat eingefügt und benutzt werden kann.

4.3.3 de.dlr.sc.virsat.model.extension.fosd.test

Dieses Package beinhaltet alle Tests für das Plugin.

4.3.4 de.dlr.sc.virsat.model.extension.fosd.ui

Dieses Package enthält alle für die UI relevanten Dateien, darunter Wizards und Pages, die die Baumstrukturen und Operationen im Repository ermöglichen.

4.4 Feature Tree

Der Feature Tree wird in der Datei *GenerateFeaturePage.java* erstellt. Mit einem Klick auf "Generate Feature Wizard" wird auf Basis des Product Trees ein Feature Tree erstellt. In Abb. 26 ist der Wizard zu sehen, der eine Vorschau der Struktur des Feature Trees gibt. "F:" kündigt ein Feature an und der darauf folgende String ist der von der PTD oder ED geerbte Name. Der Product Tree *sc* wird Knoten für Knoten kopiert und ein Cast zur Klasse *FeatureTree* durchgeführt (Listing 8).

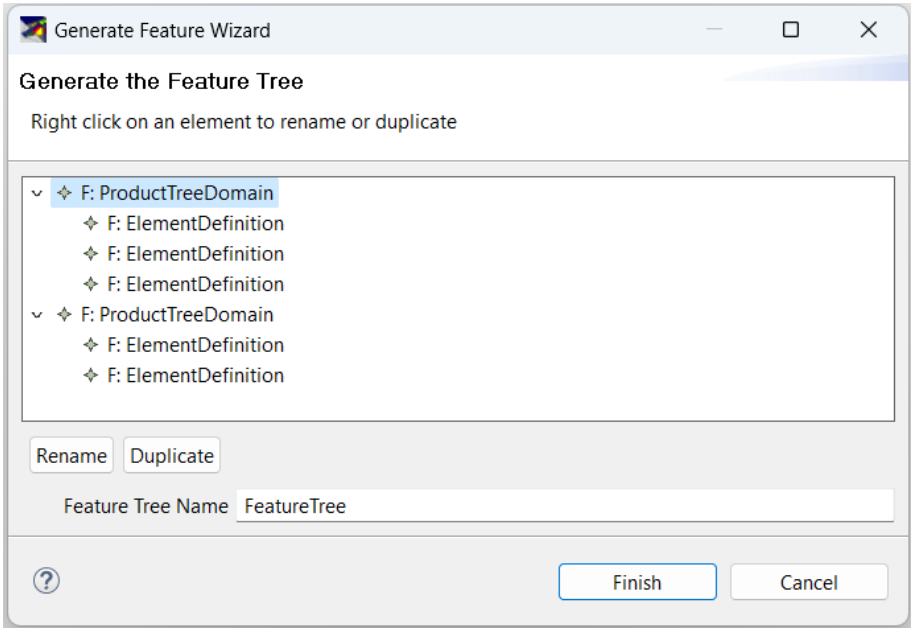


Abbildung 26: Der Wizard für das Generieren eines Feature Trees

Listing 8: Erstellen des Feature Trees

```
1 FeatureTree ct = (FeatureTree) ProductStructureHelper.createTreeModel(sc);
2 this.sei = ct.getStructuralElementInstance();
3 treeViewer.setInput(ct.getStructuralElementInstance());
```

Während des Entwurfsprozesses kann es vorkommen, dass nach Generierung des Feature Trees noch Änderungen am Product Tree vorgenommen werden. Diese Änderungen müssen selbstverständlich Teil des Feature Trees werden. Die Datei ”UpdateFeatureTree.java” ermittelt die Differenzen zwischen PT und FT und leitet entsprechende Schritte ein. Zunächst werden in Listing 9 die aus dem PT entfernten Element Definitions und Product Tree Domains gesucht. Dazu wird für jedes StructuralElement im Feature Tree überprüft, ob das *SuperSei* (ED oder PTD, von der das SE geerbt hat) noch existiert. Falls nicht, wird das SE zur Liste der entfernten Knoten hinzugefügt.

Listing 9: Finden von entfernten Element Definitions

```
1 for (StructuralElementInstance child : featureTree.getDeepChildren()) {
2         if (child.getSuperSeis().size() == 0) {
3                 removedFromProductTree.add(child);
4         }
5 }
```

Anschließend wird ermittelt, ob EDs hinzugefügt wurden. Die Größen der Knotenlisten des FTs sowie des PTs werden verglichen. Es ist wichtig, dass zu der Größe der PT-Liste die Anzahl der zuvor ermittelten, entfernten EDs addiert werden, damit nur die hinzugefügten EDs die Differenz in der Größe verursachen. Sollte die PT-Liste größer sein, werden in der *getAddedElementDefinitions()*-Methode die tatsächlich hinzugefügten EDs ermittelt. Wurden EDs zum PT hinzugefügt, werden sie anschließend auch als Features zum FT hinzugefügt. Das Eltern-SE im Feature Tree wird in der *findParentInFeatureTree()*-Methode herausgefunden,

indem das *SuperSei* des Eltern-SEs im PT gesucht wird. Dem Eltern-SE wird dann das neue Feature hinzugefügt (Listing 10).

Listing 10: Vergleich der Listengrößen

```
1  if (featureTree.getDeepChildren().size() <
2  (productTreeSEs.size() + removedFromProductTree.size())) {
3      List<StructuralElementInstance> featuresToAdd =
4      getAddedElementDefinitions(productTree.getDeepChildren(),
5      featureTree.getDeepChildren());
6
7      while (!featuresToAdd.isEmpty()) {
8                 Iterator<StructuralElementInstance> i = featuresToAdd
9                                                       .iterator();
10                while (i.hasNext()) {
11                       StructuralElementInstance featureToAdd = i.next();
12                       StructuralElementInstance parent =
13                       findParentInFeatureTree(nodesInFeatureTree,
14                       featureToAdd);
15                       if (parent == null) {
16                              continue;
17                       }
18                       addFeatureToFeatureTree(parent, featureToAdd);
19                       i.remove();
20                }
21         }
22 }
```

Sobald die addierten EDs behandelt wurden, verbleiben noch die entfernten EDs. Das Entfernen einer ED soll entweder eine *XOR*-Beziehung zwischen dem entfernten und den anderen Features der Ebene herstellen oder das Feature endgültig löschen. Deswegen muss für jedes entfernte ED im Eltern-SE nach einer *SubFeatureRelationship* gesucht werden, da im Fall einer vorhandenen SFR bereits eine Variante mit dem Feature der entfernten ED im Feature Tree definiert ist (Listing 11). Ist keine SFR vorhanden, wird die *ChangeRelationOrDeletePage* durch den Wizard aufgerufen, die dem Nutzer die Möglichkeit gibt, sich zwischen SFR und dem endgültigen Entfernen des Features zu entscheiden (Abb. 27).

Listing 11: Entfernte EDs ermöglichen dem Nutzer eine Wahl

```
1  for (StructuralElementInstance sei : removedFromProductTree) {
2         Optional<CategoryAssignment> subFeatureRelationship =
3         sei.getParent().getCategoryAssignments().stream()
4             .filter(ca -> ca.getType().getName()
5             .equals("SubFeatureRelationship"))
6             .findAny();
7         if (subFeatureRelationship.isEmpty()) {
8         Shell shell = HandlerUtil.
9         getActiveWorkbenchWindow(event).getShell();
```

```
10          ChangeRelationOrDeleteWizard wizard =
11          new ChangeRelationOrDeleteWizard();
12          wizard.setSei(sei);
13          WizardDialog dialog = new WizardDialog(shell, wizard);
14          dialog.open();
15          }
16  }
```

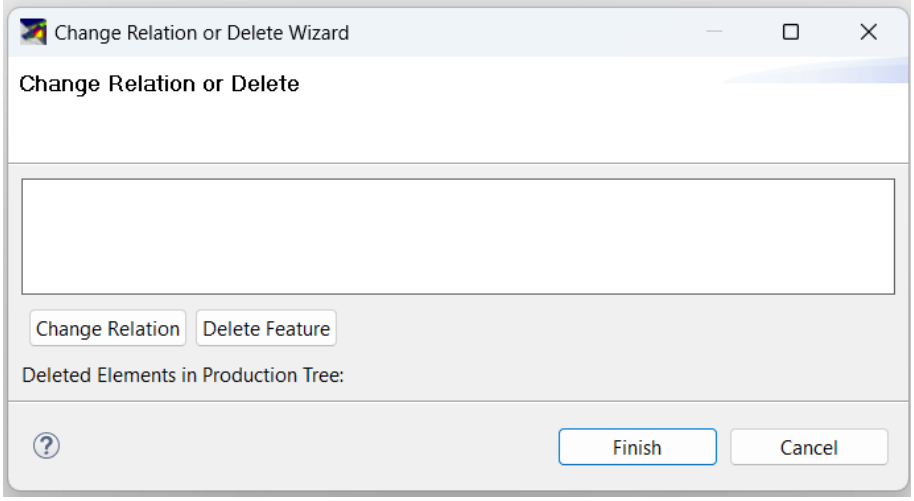


Abbildung 27: Nach dem Entfernen einer ED kann sich zwischen einer *XOR*-Beziehung und dem endgültigen Löschen entschieden werden.

4.5 Auswertung

Wenn der Ingenieur an einem Punkt ist, an dem er die Varianten sichten und sich ggf. für eine erntscheiden möchte, kann er mit einem Klick auf ”Generate Variant Wizard” das Feature-Modell aus dem Feature Tree erstellen und die Analyse des Modells in Gang setzen. Daraufhin werden die Knoten des Feature Trees auf ihre hierarchischen Beziehungen und Attribute untersucht. Dazu wird in der Datei ”VariantSelectionPage.java” die Methode *getAllValidVariants()* aufgerufen. Der Kern der Methode ist die Übertragung des Feature-Modells in die konjunktive Normalform (CNF) im DIMACS-Format, um dieses Problem anschließend an einen SAT-Solver zu geben. Die CNF baut auf der Grundlage auf, dass alle Klauseln mit einem logischen AND verknüpft sind. Die einzelnen Variablen sind mit einem logischen OR verknüpft. Eine Variable kann mit einem logischen NOT negiert werden. Eine beispielhafte bool’sche Aussage ist in Gleichung 5 dargestellt (Beispiel entnommen aus [15]).

$$\begin{aligned}&(x_1 \; OR \; (\; NOT \; x_3 \;) \;) \; AND \\ &(x_2 \; OR \; x_3 \; OR \; (\; NOT \; x_1 \;) \;)\end{aligned} \tag{5}$$

Das DIMACS-Format nimmt diese Regeln der Verknüpfung von Klauseln und Variablen implizit an, wodurch die explizite Nennung der Operatoren entfällt. Dabei folgt die DIMACS-Datei dem ASCII-Format und hat feste Regeln [15]:

1. Kommentare werden mit dem kleinen Buchstaben ”c” eingeleitet und können überall

in der Datei auftauchen. Üblicherweise erfolgt dies aber zu Beginn.

2. Anschließend folgt die Zeile, die das Problem aufstellt. Sie wird mit dem kleinen ”p” eingeleitet und enthält den Problemtypen, also ”cnf”, die Anzahl der Variablen und die Anzahl der Klauseln.

3. Die restlichen Zeilen definieren nacheinander die Klauseln. Eine Klausel listet die Variablen hintereinander auf, wobei der Operator OR sich implizit zwischen den Variablen befindet. Ein NOT Operator wird mit einer negativen Variable indiziert. Die Variablen nehmen also den Zahlenraum der ganzen Zahlen exklusive der Null ein.

4. Eine Klausel kann sich über mehrere Zeilen erstrecken.

5. Das Ende einer Klausel wird mit ”0” indiziert.

Die bool’sche Aussage aus Gleichung 5 nimmt im DIMACS-Format die Form in Listing 12 an. Es sind drei Variablen zu sehen, wobei das Mapping $x_1 = 1, x_2 = 2, x_3 = 3$ zwischen den Variablen aus der bool’schen Aussage und den DIMACS-Variablen gilt. Weiterhin sind zwei Klauseln zu sehen, die jeweils die OR-Verknüpfungen aus der bool’schen Aussage repräsentieren und mit ”0” abgeschlossen werden.

Listing 12: DIMACS-Format

```
c
p cnf 3 2
1 -3 0
2 3 -1 0
```

Da die Variablen mit Zahlen aus dem Raum $\mathbb{Z}$ ohne Null benannt werden, muss zunächst ein bidirektionales Mapping zwischen der Variablen aus der CNF und den Features aus dem Feature-Modell hergestellt werden.

Listing 13: Mapping der DIMACS-Variablen und Feature-Objekte

```
1
2  // put root feature
3  variableToFeature.put(1, featureTree);
4  featureToVariable.put(featureTree, 1);
5
6  // next variable is number 2
7  int variableCounter = 2;
8
9  for (StructuralElementInstance feature :
10     featureTree.getDeepChildren()) {
11         variableToFeature.put(variableCounter, feature);
12         featureToVariable.put(feature, variableCounter);
13         variableCounter++;
14 }
```

Um Effizienz zu gewährleisten, machen wir uns die durchschnittliche Zugriffszeit von $O(1)$ einer unidirektionalen Hashmap zunutze und befüllen deswegen zwei separate Hashmaps;

eine für das Mapping von DIMACs-Variable zu Feature-Objekt und eine für das Mapping von Feature-Objekt zu DIMACS-Variable. Dazu wird zunächst das Wurzel-Feature *Feature Tree* mit dem Wert "1" und anschließend die Liste der Features aus dem Wiedergabewert der Methode *getDeepChildren()* des Feature Trees mit aufsteigender Zahlenfolge gemappt (Listing 13).

Listing 14: ModelIterator mit dem Allrounder-SAT-Solver

```
1 /*
2 * Use ModelIterator to find all valid variants.
3 */
4 ISolver solver = new ModelIterator(SolverFactory.newDefault());
```

Zum Lösen der CNF-Klauseln wird der *default* SAT-Solver aus der SAT4J-Library genutzt. Ein SAT-Solver arbeitet normalerweise nur so lange bis die erste Lösung gefunden wurde. Doch da die Variantenselektion offensichtlich nur möglich ist, sofern alle validen Varianten des Feature-Modells gefunden und dem Nutzer präsentiert werden können, darf der SAT-Solver erst aufhören, sobald alle Lösungen gefunden wurden. Dafür wird der ModelIterator aus der SAT4J-Library benutzt (Listing 14), der sich gefundene Lösungen merkt und erst abbricht, wenn er nach mehreren Iterationen keine neuen Lösungen mehr findet.
Anschließend werden die CategoryAssignments und hierarchischen Beziehungen für jedes Feature des Feature-Modells betrachtet und in CNF-Klauseln übertragen. Hier ist zu beachten, dass die SubFeatureRelationship (*XOR, OR*) von dem jeweiligen Eltern-Feature betrachtet wird, damit die anderen Kind-Features auf dieser Ebene gefunden werden können.

Listing 15: SubFeatureRelationship vom Eltern-Feature wird zu einer CNF-Klausel der Kind-Features gemacht

```
1  /*
2  * Handle SubFeatureRelationship.
3  */
4  if (parentSubFeatureRelationship.isPresent()) {
5
6          SubFeatureRelationship subFeatureRelationship =
7          new SubFeatureRelationship(
8              parentSubFeatureRelationship.get()
9          );
10
11         if (subFeatureRelationship.getCharacter() != null &&
12         !existingSubFeatureRelationships.contains(feature.getKey()
13         )) {
14
15         /* For SubFeatureRelationships we need the
16         * other affected feature variables.
17         */
18         int[] subFeatureVariables = parent.getChildren().stream()
19                 .mapToInt(child -> featureToVariable.get(child))
```

```
20                .toArray();
21
22          .
23          .
24          .
25
26          switch(subFeatureRelationship.getCharacter()) {
27            case "xor" -> handleXor(subFeaturePlusParent, solver);
28            case "or" -> handleOr(subFeaturePlusParent, solver);
29          }
30          existingSubFeatureRelationships.addAll(
31              IntStream.of(subFeatureVariables)
32              .boxed()
33              .collect(Collectors.toList()));
34          }
35  }
```

Listing 15 beinhaltet die Logik der Behandlung von SubFeatureRelationships im Eltern-Feature. Zeile 9 und 10 prüfen, ob der Typ der SubFeatureRelationship definiert wurde und ob das aktuell überprüfte Kind-Feature schon in einer bereits behandelten SubFeatureRelationship aufzufinden ist, damit Klauseln nicht doppelt erzeugt werden. Danach werden in Zeile 15-17 die anderen Kind-Features dieser Ebene geholt und deren CNF-Variablen in einem Array gespeichert. Die CNF-Variablen der Kind-Features und des Eltern-Features werden in Zeile 26-29 an die entsprechende Methode übergeben, die dann die *XOR*- oder *OR*-Klausel erstellt. Die in den erstellten Klauseln enthaltenen Kind-Features werden abschließend zur Liste der bereits bearbeiteten SubFeatureRelationships hinzugefügt.
Eine *XOR*- oder *OR*-Beziehung zwischen den Kind-Features eines Eltern-Features überschreibt jede Beziehung, die ein einzelnes Kind-Feature zu dem Eltern-Feature haben könnte, d.h. ein Kind-Feature, das in einer *XOR-/OR*-Beziehung vorkommt, kann zwar im Kontext dieser Beziehung optional sein, aber niemals eine eins-zu-eins *Optional*-Beziehung mit dem Eltern-Feature haben. Deswegen wird das CategoryAssignment *OptionalRelationship* nur behandelt, falls keine *SubFeatureRelationship* beim Eltern-Feature vorzufinden ist. Wenn keine *SubFeatureRelationship* vorliegt, eine *OptionalRelationship* im aktuell zu untersuchenden Feature vorzufinden ist und diese den Wert ”true” hat, werden die CNF-Variable des Eltern-Features und des aktuell zu untersuchenden Features an die *handleOptional()*-Methode übergeben, die die Klauseln erstellt. Sollte keine *OptionalRelationship* vorliegen, der Wert auf ”false” gesetzt oder gar nicht deklariert sein, werden die o.g. CNF-Variablen an die *handleMandatory()*-Methode übergeben, die für diesen Fall die Klauseln erstellt.

Listing 16: CrossTreeConstraint wird untersucht

```
1  .
2  .
3  .
4  // Get other feature variable by uuid
5  int referencedFeatureVariable = 0;
```

```
6  for (Entry<Integer, StructuralElementInstance> otherFeature
7  : variableToFeature.entrySet()) {
8      if (otherFeature.getValue().getUuid().toString()
9      .equals(crossTreeConstraintInstance.getReferenceUuid())) {
10         referencedFeatureVariable = otherFeature.getKey();
11     }
12 }
13
14 switch (crossTreeConstraintInstance.getCharacterBean()
15 .getValue()) {
16 // Requires
17 case "enumValue1" -> handleRequires(referencedFeatureVariable,
18                                     feature.getKey(), solver);
19 // Excludes
20 case "enumValue2" -> handleExcludes(referencedFeatureVariable,
21                                     feature.getKey(), solver);
22 }
```

Bevor die Klauseln durch den SAT-Solver auf Lösungen untersucht werden, wird noch das CategoryAssignment *CrossTreeConstraint* überprüft. Falls eine Instanz im untersuchten Feature vorliegen sollte, wird über die anderen Features im Feature-Modell iteriert und das Feature gesucht, dessen UUID mit der im *CrossTreeConstraint* gespeicherten *ReferenceUuid* übereinstimmt. Die CNF-Variablen dieses Features und des aktuell untersuchten Features werden je nach Typ des Constraints entweder an die *handleRequires()*- oder an die *handleExcludes()*-Methode übergeben (Listing 16).

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2),\ mit\ x_1\ als\ Eltern-Feature\ und\ x_2\ als\ Kind-Feature. \quad (6)$$

Nehmen wir die in Kapitel 2.2.1 beschriebene Methode zur Überführung einer *Mandatory*-Beziehung in die CNF, erhalten wir Gleichung 6.

Listing 17: *handleMandatory()*-Methode

```
1 solver.addClause(new VecInt(
2 new int[]{parentVariable,childVariable})).toString();
3 solver.addClause(new VecInt(
4 new int[]{-childVariable, parentVariable})).toString();
```

Die *handleMandatory()*-Methode nimmt die CNF-Variablen des Eltern- und des Kind-Features und erstellt daraus zwei Vektoren, die jeweils eine Klausel mit einer negierten und einer nicht negierten Variable enthält (Listing 17). Diese Vektoren entsprechen der DIMACS-Semantik und würden in einer DIMACS-Datei wie in Listing 18 aussehen (mit 1 = Eltern-Feature und 2 = Kind-Feature. Die Klauseln werden dann dem SAT-Solver hinzugefügt.

Listing 18: *Mandatory*-Beziehung im DIMACS-Format

```
c
p cnf 2 2
1 -2 0
```

```
-2 1 0
```

Die *handleOptional()*-Methode nimmt ebenfalls die CNF-Variablen des Eltern- und Kind-Features, erstellt daraus aber nur einen Vektor, der die Kind-Variable negiert (Listing 19). In Dimacs-Notation wäre diese Klausel "-2 1 0".

Listing 19: *handleOptional()*-Methode

```
solver.addClause(new VecInt(new int[]{-childVariable, parentVariable}));
```

Eine Requires-Beziehung hat den selben Charakter wie eine *Optional*-Beziehung, d.h. eine optionale Variable ist von einer obligatorischen Variable abhängig. Deswegen ruft *handleRequires()* lediglich die *handleOptional()*-Methode auf, die die optionale Variable negiert. Eine *Excludes*-Beziehung bezweckt, dass die betroffenen Variablen nicht den gleichen Wert haben dürfen, d.h. es dürfen nicht beide fehlen und nicht beide vorkommen. Dafür erstellt die *handleExcludes()*-Methode eine Klausel, die beide Variablen negiert (Listing 20).

Listing 20: *handleExcludes()*-Methode

```
solver.addClause(new VecInt(new int[]{-excludedVariable, -thisVariable}));
```

Eine *OR*-Beziehung zwischen zwei Variablen 2 und 3 und einer Eltern-Variable 1 sieht in DIMACS-Notation wie in Listing 21 aus.

Listing 21: *OR*-Beziehung im DIMACS-Format

```
c
p cnf 3 3
2 3 -1 0
-2 1 0
-3 1 0
```

Die *handleOr()*-Methode nimmt ein Array der Kind-Variablen mit der Eltern-Variable an letzter Stelle, negiert diese letzte Stelle, verpackt sie in einen Vektor und fügt die Klausel dem Solver hinzu. Anschließend wird für jede Kind-Variable jeweils noch eine Klausel mit negierter Kind-Variable erstellt (Listing 22).

Listing 22: *handleOr()*-Methode

```
1  try {
2          int[] copy = subFeatures.clone();
3          copy[copy.length-1] = -copy[copy.length-1];
4          IVecInt v = new VecInt(copy);
5          solver.addClause(v);
6  } catch (ContradictionException e1) {
7          e1.printStackTrace();
8  }
9
10 for (int i = 0; i < children.length; i++) {
11         try {
12                 IVecInt v = new VecInt(new int[]{-children[i], parent});
13                 solver.addClause(v);
```

```
14          } catch (ContradictionException e) {
15                  e.printStackTrace();
16      }
17  }
```

4.6 Variantenselektion und Configuration Tree

Nachdem alle Klauseln erzeugt und dem Solver in der *getAllValidVariants*-Methode hinzugefügt wurden, kann nun das eigentliche Auflösen des Feature-Modells in die validen Varianten folgen (Listing 23). Der ModelIterator hat die Methode *isSatisfiable()* (Zeile 3), mit der überprüft wird, ob noch Lösungen für das Problem zu finden sind. Deswegen lassen wir den Solver solange neue Lösungen für das Problem erstellen und zu einer Liste hinzufügen wie noch nicht gefundene Lösungen existieren. Der ModelIterator erkennt selbstständig bereits behandelte Lösungen und gibt sie nicht mehr aus. Eine Lösung wird auch als Modell bezeichnet und besteht aus einem IntegerArray, das die Variablen des Feature-Modells beinhaltet. Variablen, die in einem Modell, d.h. also in einer Variante nicht vorkommen, sind negativ. Sind alle Modelle gefunden, wird noch das an erster Stelle aus der Liste entfernt (Zeile 7), da es der Fall ist, in dem alle Variablen negativ und damit nicht Teil des Feature-Modells sind.

Listing 23: Der Solver findet Lösungen

```
1  boolean unsat = true;
2  List<int[]> models = new ArrayList<>();
3  while (solver.isSatisfiable()) {
4      unsat = false;
5      models.add(solver.model());
6  }
7  models.remove(0);
```

Zuletzt laufen alle Modelle nochmal durch einen Filter, der überprüft, ob das externe Constraint *MassBudget* eingehalten wird. Dazu wird über die positiven Variablen eines Modells iteriert und die Masse des jeweiligen Features zur Gesamtmasse der Variante addiert. Varianten, die das *MassBudget* überschreiten, werden aus der Liste der validen Varianten entfernt.
Die validen Varianten werden in der *createTreeUI*-Methode zu Configuration Trees verarbeitet, aus denen der Nutzer dann Varianten auswählen kann. In Listing 24 wird als erstes ein Configuration Tree auf Basis des gesamten Feature Trees instanziiert. Da wir aber nur die Variablen der validen Variante als Element Configuration zum CT hinzufügen wollen, entfernen wir alle SEs aus dem CT. Lediglich das Wurzel-Feature bzw. das SE *ConfigurationTree* ist noch Teil des CTs. Der *featureToElementConfigurationMapper* erlaubt uns das auffinden der zum CT hinzugefügten ECs anhand der Variablen des Features aus dem FT. Das Wurzel-Feature wird direkt zum Mapper hinzugefügt.

Listing 24: Ein leerer Configuration Tree

```
1  // Create a configurationTree instance with repository as parent.
2  ConfigurationTree ct = (ConfigurationTree)
```

```
3       ProductStructureHelper.createTreeModel(sc);
4   // We merely want to add the features of the specific configuration.
5   ct.removeAllStructuralElementInstance(
6       ct.getDeepChildren(IBeanStructuralElementInstance.class));
7   StructuralElementInstance rootFeature = ct.getStructuralElementInstance();
8   featureToElementConfigurationMapper.put(sc, rootFeature);
```

Danach werden alle Variablen der Lösung durchlaufen und zu jeder positiven Variable eine EC auf Basis des Features im Feature Tree instanziiert und zum Eltern-SE im Configuration Tree hinzugefügt. Sind alle Varianten als Configuration Tree instanziiert, werden sie einem TreeViewer als Input übergeben und in einem Wizard dargestellt (Abb. 28). Der Nutzer kann im Wizard eine Variante auswählen, die dann umgehend als Configuration Tree im Repository erzeugt wird.

Abbildung 28: Der Wizard für das Selektieren und Generieren einer Variante

5 Evaluation

In diesem Kapitel soll die erzielte Funktionalität anhand eines Beispiels erprobt werden. Anschließend werden die Vorteile gegenüber dem iterativen Entwurfsprozess in Virtual Satellite erklärt und wo die Grenzen des aktuellen Prototypen liegen. Mit dem gewonnenen Wissen werden dann die Forschungsfragen beantwortet.

5.1 Beispiel

Das folgende Beispiel fügt dem Satellitenentwurf Schritt für Schritt Subsysteme und Komponenten hinzu. Dabei ist für jeden Schritt der Product Tree, der daraus generierte Feature Tree und das aus dem Feature Tree abgeleitete Feature-Modell dargestellt.

5.1.1 AOCS

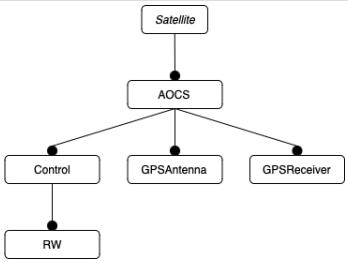
Product Tree	Feature Tree	Feature-Modell
PT: ProductTree documents PTD: AOCS documents PTD: Control documents ED: RW ED: GPSAntenna ED: GPSReceiver	FT: FeatureTree documents F: AOCS documents F: Control documents F: RW F: GPSAntenna F: GPSReceiver	

Tabelle 3: Erstellung des AOCS

Das Attitude and Orbit Control System (*AOCS*) wird als Product Tree Domain im Product Tree erstellt. Darunter werden die Ausrichtungskomponenten in der PTD *Control* hinzugefügt. Zunächst wird ein Reaction Wheel (*RW*) ausgewählt. Weiterhin werden die Komponenten *GPSAntenna* und *GPSReceiver* als Element Definitions unter dem *AOCS* erstellt. Der Feature Tree übernimmt die Struktur und lässt die Features von den PTDs und EDs erben. Das Feature-Modell besteht aus dem abstrakten Wurzel-Feature *Satellite* und fünf obligatorischen Features.

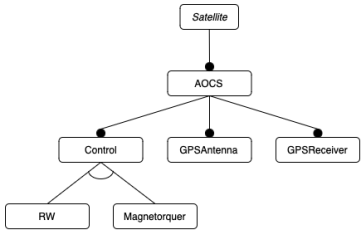
Product Tree	Feature Tree	Feature-Modell
PT: ProductTree documents PTD: AOCS documents PTD: Control documents ED: Magnetorquer ED: GPSAntenna ED: GPSReceiver	FT: FeatureTree documents F: AOCS documents F: Control documents F: RW F: Magnetorquer SFR: SubFeatureRelationship character: xor=XOR F: GPSAntenna F: GPSReceiver	

Tabelle 4: RW wird im Product Tree entfernt

Wir überlegen das Reaction Wheel gegen einen Magnetorquer auszutauschen, da die geplante Mission im unteren Orbit stattfindet und wir so optimal das Erdmagnetfeld ausnutzen können. Wir entfernen das Reaction Wheel aus dem Product Tree und fügen einen Magnetorquer hinzu. Wir werden gefragt, ob wir das Reaction Wheel endgültig löschen wollen oder die Beziehung zu *XOR* ändern wollen. Wir ändern zu *XOR*. Im Product Tree ist nur noch der Magnetorquer zu sehen, aber da der Feature Tree den Prozess dokumentiert, sind hier noch beide Komponenten vorzufinden. Eine SubFeatureRelationship des Typs *XOR* wurde auf Ebene der beiden Komponenten erstellt. Im Feature-Modell wird diese Beziehung ebenfalls abgebildet.

5.1.2 TCS

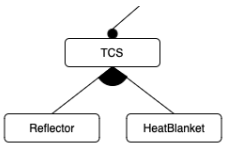
Product Tree	Feature Tree	Feature-Modell
PT: ProductTree documents PTD: AOCS documents PTD: Control documents ED: Magnetorquer ED: GPSAntenna ED: GPSReceiver PTD: TCS documents ED: HeatBlanket ED: Reflector	FT: FeatureTree documents F: AOCS documents F: Control F: GPSAntenna F: GPSReceiver F: TCS documents F: Reflector F: HeatBlanket SFR: SubFeatureRelationship character: or=OR	

Tabelle 5: TCS wird hinzugefügt

Nun erstellen wir das Temperature Control System (*TCS*) als PTD im Product Tree. Wir überlegen, ob eine Heizdecke oder ein Reflektor ausreicht oder, ob wir beides benötigen. Im Feature Tree dokumentieren wir diese Überlegung mit einer SubFeatureRelationship des Typs *OR*. Im Feature-Modell wird die Beziehung zwischen den Kind-Features abgebildet.

5.1.3 EPS

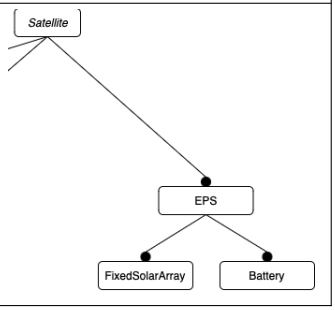
Product Tree	Feature Tree	Feature-Modell
PT: ProductTree documents PTD: AOCS PTD: TCS PTD: EPS documents ED: FixedSolarArray ED: Battery	FT: FeatureTree documents F: AOCS F: TCS F: EPS documents F: FixedSolarArray F: Battery	

Tabelle 6: Hinzufügen des EPS

Das Energy Production System (*EPS*) erstellen wir gleichfalls als PTD und fügen ein *FixedSolarArray* sowie eine *Battery* als EDs hinzu. Feature Tree und Feature-Modell stellen diese Änderungen analog als normale obligatorische Features dar.

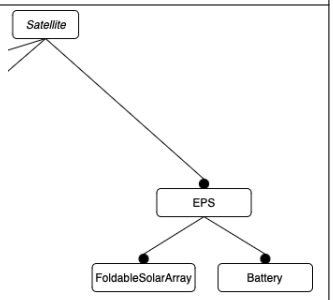
Product Tree	Feature Tree	Feature-Modell
PT: ProductTree documents PTD: AOCS PTD: TCS PTD: EPS documents ED: Battery ED: FoldableSolarArray	FT: FeatureTree documents F: AOCS F: TCS F: EPS documents F: Battery F: FoldableSolarArray	

Tabelle 7: Endgültiges Entfernen des FixedSolarArray

Wir entscheiden uns gegen das *FixedSolarArray* und wollen es gegen ein *FoldableSolarArray* tauschen. Beim Löschen des *FixedSolarArray* werden wir wieder gefragt und wir Klicken auf das endgültige Löschen. So taucht es in keinem der drei Bäume mehr auf und ist final gegen das *FoldableSolarArray* getauscht.

5.1.4 OBC

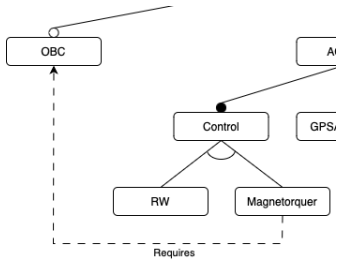
Product Tree	Feature Tree	Feature-Modell
PT: ProductTree documents PTD: AOCS PTD: TCS PTD: EPS PTD: OBC documents	F: AOCS documents F: Control documents F: RW F: Magnetorquer documents CTC: CrossTreeConstraint character: enumValue1=REQUIRES referenceUuid: 4813bebb-4ad5-4d SFR: SubFeatureRelationship F: GPSAntenna F: GPSReceiver F: TCS F: EPS F: OBC documents OR: OptionalRelationship	

Tabelle 8: Optionales OBC mit Cross-Tree-Constraint

Da wir uns in einer frühen Entwurfsphase befinden, sind wir uns noch nicht sicher, ob und in welcher Form wir einen On-Board-Computer (*OBC*) benötigen. So erstellen wir eine Product Tree Domain *OBC* und dokumentieren unsere Unsicherheit als OptionalRelationship im Feature Tree. Ein Magnetorquer benötigt allerdings auf jeden Fall einen *OBC* und wir erstellen einen CrossTreeConstraint im Feature Tree unter dem Feature *Magnetorquer*. Der Typ ist *REQUIRES* und wir referenzieren den *OBC* mit seiner UUID. Im Feature-Modell ergibt sich somit ein gestrichelter Pfeil vom Magnetorquer-Feature, der den *Requires*-Cross-Tree-Constraint zum optionalen OBC-Feature ausdrückt.

5.1.5 Structure

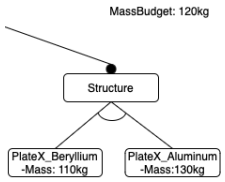
Product Tree	Feature Tree	Feature-Modell
PTD: Structure documents ED: PlateX_Aluminum documents ME: MassEquipment EquationSection margin: 20.0% mass: 130kg massMargin: 26.0kg massWithMargin: 156.0kg ED: PlateX_Beryllium documents ME: MassEquipment EquationSection margin: 20.0% mass: 110kg massMargin: 22.0kg massWithMargin: 132.0kg	F: Structure documents F: PlateX_Beryllium F: PlateX_Aluminum SFR: SubFeatureRelationship character: xor=XOR MB: MassBudget mass: 120kg	

Tabelle 9: Massenattribute und MassBudget

Im Product Tree erstellen wir noch eine PTD *Structure* und fügen ihr zwei Möglichkeiten für die horizontale Struktur *PlateX* hinzu. Eine besteht aus Aluminium und eine aus Beryllium. Der aus Aluminium fügen wir ein Attribut *MassEquipment* mit einer Masse von 130kg und der aus Beryllium eins mit der Masse 110kg hinzu. Die Features im Feature Tree erben diese Attribute von den EDs und werden mit einer *XOR*-SubFeatureRelationship verknüpft. Außerdem kennen wir jetzt die maximale Masse, die der Satellit beim Start haben darf und fügen diese Grenze von 120kg als dem Feature Tree als externes Constraint *MassBudget* hinzu. Im Feature-Modell erhalten die Kind-Features des Structure-Features ihre jeweiligen Attribute und eine *XOR*-Beziehung. Das externe Constraint ist kein direkter Teil des Feature-Modells.

5.1.6 Varianten

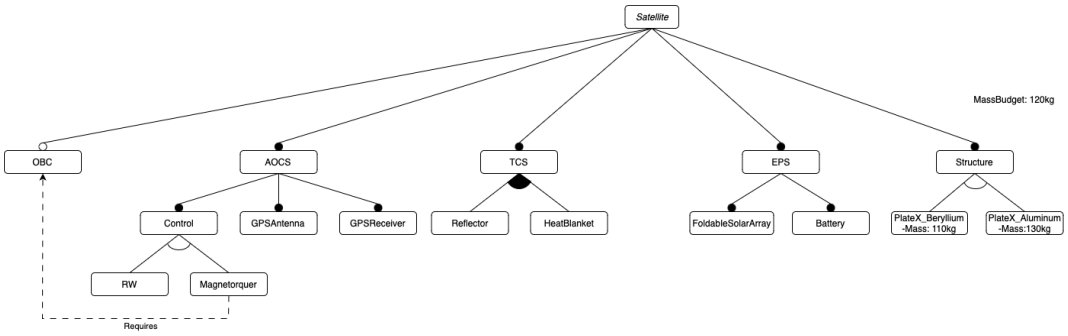


Abbildung 29: Das gesamte Feature-Modell des Satellitenentwurfs

Das gesamte Feature-Modell in Abb. 29 stellt den kompletten Variantenraum ab. Dieses Modell kann nun ausgewertet und alle Varianten des Entwurfsprozess extrahiert werden. Das machen wir im FOSD-Plugin, indem wir ”Generate Variant Wizard” klicken. In den

Tabellen 10 bis 12 sind alle gefundenen Varianten und die zugehörige Konfiguration als Feature-Modell aufgelistet. Eine oder mehrere Varianten können ausgewählt und dann als Configuration Tree instanziiert werden.

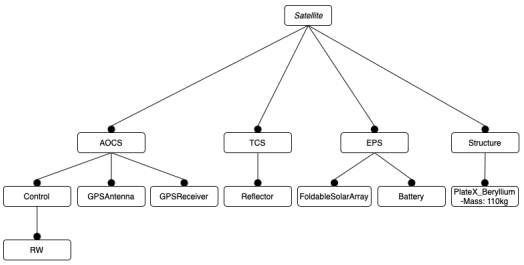
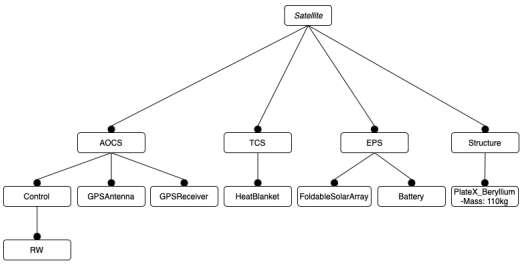
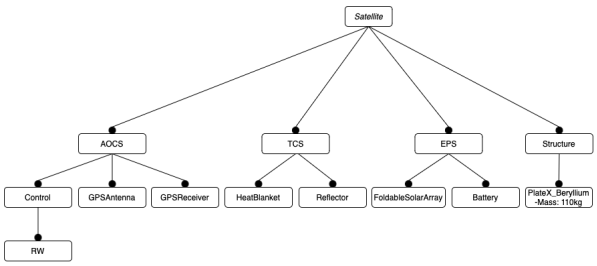
Variante	Feature-Modell
EC: AOCS EC: Control EC: RW EC: GPSAntenna EC: GPSReceiver EC: TCS EC: Reflector EC: EPS EC: FoldableSolarArray EC: Battery EC: Structure EC: PlateX_Beryllium	
EC: AOCS EC: GPSReceiver EC: Control EC: RW EC: GPSAntenna EC: TCS EC: HeatBlanket EC: EPS EC: FoldableSolarArray EC: Battery EC: Structure EC: PlateX_Beryllium	
EC: EPS EC: FoldableSolarArray EC: Battery EC: TCS EC: HeatBlanket EC: Reflector EC: AOCS EC: GPSReceiver EC: Control EC: RW EC: GPSAntenna EC: Structure EC: PlateX_Beryllium	

Tabelle 10: Alle Varianten und die Konfigurationen als Feature-Modell (1/3)

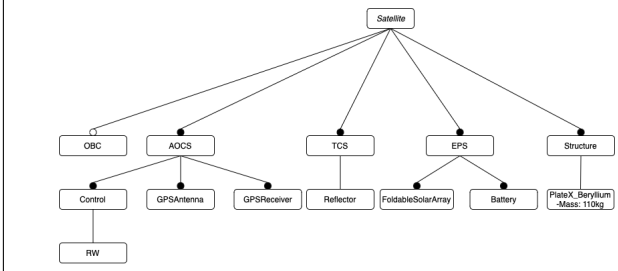
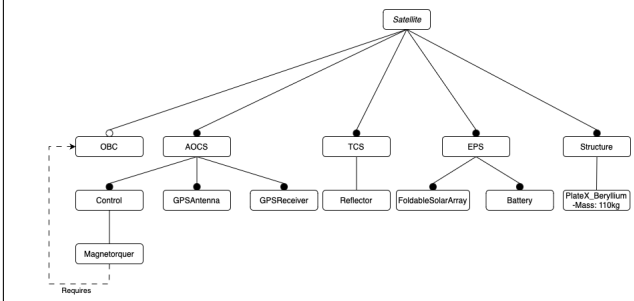
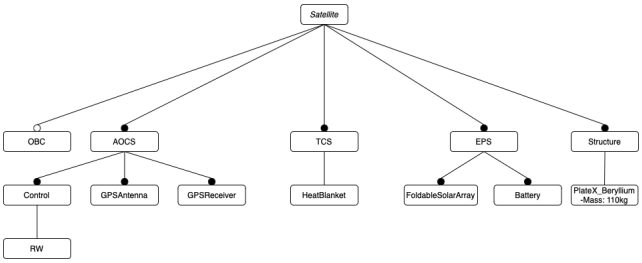
Variante	Partielle Konfiguration
EC: OBC EC: EPS EC: Battery EC: FoldableSolarArray EC: AOCS EC: GPSReceiver EC: Control EC: RW EC: GPSAntenna EC: Structure EC: PlateX_Beryllium EC: TCS EC: Reflector	
EC: AOCS EC: Control EC: Magnetorquer EC: GPSReceiver EC: GPSAntenna EC: TCS EC: Reflector EC: Structure EC: PlateX_Beryllium EC: OBC EC: EPS EC: Battery EC: FoldableSolarArray	
EC: EPS EC: FoldableSolarArray EC: Battery EC: OBC EC: AOCS EC: GPSReceiver EC: Control EC: RW EC: GPSAntenna EC: TCS EC: HeatBlanket EC: Structure EC: PlateX_Beryllium	

Tabelle 11: Alle Varianten und die Konfigurationen als Feature-Modell (2/3)

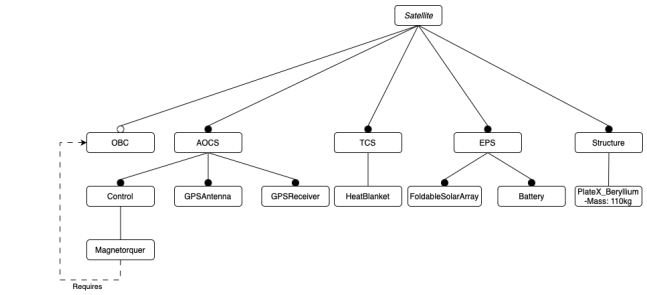
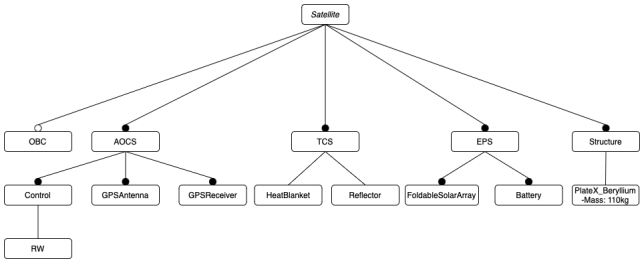
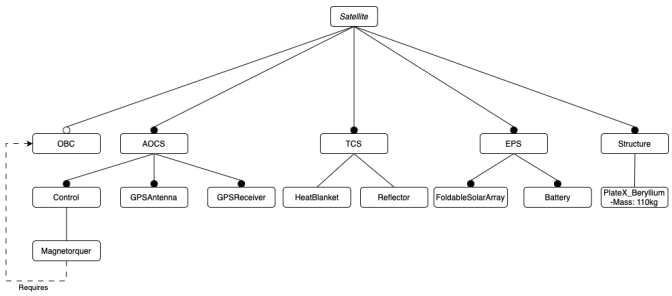
Variante	Partielle Konfiguration
EC: TCS EC: HeatBlanket EC: OBC EC: Structure EC: PlateX_Beryllium EC: EPS EC: Battery EC: FoldableSolarArray EC: AOCS EC: GPSReceiver EC: GPSAntenna EC: Control EC: Magnetorquer	
EC: AOCS EC: GPSReceiver EC: GPSAntenna EC: Control EC: RW EC: OBC EC: EPS EC: FoldableSolarArray EC: Battery EC: Structure EC: PlateX_Beryllium EC: TCS EC: HeatBlanket EC: Reflector	
EC: Structure EC: PlateX_Beryllium EC: OBC EC: TCS EC: Reflector EC: HeatBlanket EC: EPS EC: FoldableSolarArray EC: Battery EC: AOCS EC: Control EC: Magnetorquer EC: GPSReceiver EC: GPSAntenna	

Tabelle 12: Alle Varianten und die Konfigurationen als Feature-Modell (3/3)

5.2 Bewertung

Im obigen Beispiel wurden neun mögliche Variante bzw. Konfigurationen gefunden, aus denen der Nutzer auswählen kann. Um das erzielte Ergebnis zu prüfen, wurde das Feature-Modell aus Abb. 29 einer FeatureIDE-Instanz ([3]) übergeben. Dabei war zu beachten, dass durch das externe Constraint *MassBudget* das Feature *PlateX_Aluminum* nie ausgewählt werden kann, weswegen es bei der Übertragung in die FeatureIDE nicht beachtet wurde. Die Beziehung zwischen dem Feature *Magnetorquer* und *OBC* wurde als Constraint in der Form «Magnetorquer implies OBC» definiert. Anschließend wurde mit dem Product Generator der FeatureIDE auf alle möglichen Konfigurationen untersucht. Das Analyseergebnis ist in Abb. 30 zu sehen.

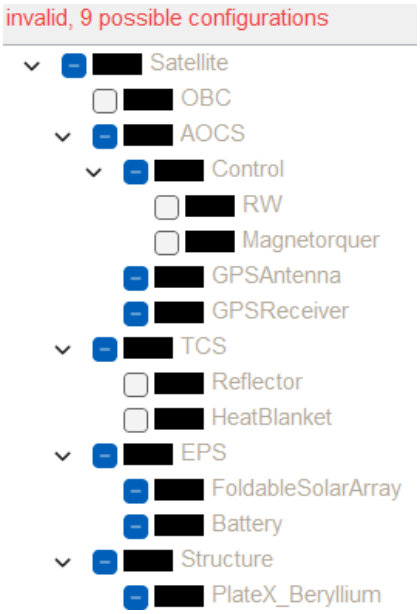


Abbildung 30: Das Ergebnis der FeatureIDE bei Instanziierung des Beispiels

Die FeatureIDE gibt an erster Stelle eine Konfiguration aus, in der alle Features selektiert sind, die immer selektiert werden würden, da sie durch eine *Mandatory*-Beziehung oder durch Kombination der logischen Operatoren immer ”true” sein müssen. In dieser Konfiguration kann dann neben den noch nicht ausgewählten Features ein Haken gesetzt werden, um sie in die Konfiguration aufzunehmen. Sobald eine Konfiguration valide ist, ändert sich das ”invalid” oben zu ”valid”. Insgesamt hat die FeatureIDE neun valide Konfigurationen gefunden, was mit dem Ergebnis des obigen Beispiels übereinstimmt. Auch beim Gegenprüfen (Vergleich in Anhang A.1) der einzelnen Konfigurationen mit denen aus dem Beispiel wird deutlich, dass sie übereinstimmen, d.h. es wurden alle möglichen Varianten des Entwurfsprozesses gefunden. Damit kann auch die konzeptionelle Anforderung K4 bestätigt werden. Sie besagt, dass das Feature-Modell die Menge aller möglichen Varianten des Satellitenentwurfsprozesses abbilden muss. Die Prüfung des Feature-Modells mittels einer externen Software hat diese Funktionalität bestätigt. Der Ingenieur ist damit in der Lage, alte Varianten zu behalten, Entscheidung zu verschieben und dann auf Basis von eigens definierten Constraints alle noch validen Varianten, die während des Entwurfsprozesses entstanden sind, automatisiert ausgeben zu lassen. Jede dieser Varianten kann er als Configuration Tree instanziieren und den gewohnten Arbeitsablauf aus Virtual Satellite fortsetzen.

Wie jede prototypisch implementierte Sofware hat auch diese ihre Grenzen. Wie in der Software Requirements Specification in Kapitel 3.1 beschrieben, wurde aufgrund des zeitlichen Rahmens eine elaborierte Nutzeroberfläche ausgelassen. Es wurden schon bestehende Wizards des Product-Structure-Plugins verwendet, um die Elemente graphisch anzuzeigen. Der Wizard für die Variantenselektion z.B. ordnet die gefundenen Varianten vertikal an, statt sie horizontal darzustellen, sodass bei einer großen Anzahl (ca. >5) gefundener Varianten oder bei kleinen Bildschirmen einige Varianten hinter dem Bildschirmrand verschwinden. Ein weiteres graphisches Problem taucht im Wizard auf, der den Nutzer bei Entfernen einer Element Definition aus dem Product Tree fragt, ob die Beziehung geändert oder das Feature gelöscht werden soll. Hier wird nicht angezeigt, um welche Element Definition es sich handelt,

was vor allem bei mehreren gleichzeitig gelöschten EDs zum Nachteil wird. Weiterhin wurde auf Kardinalitäten verzichtet, die für den Satellitenentwurf durchaus wichtig sein können, da manche Bauteile öfter vorkommen. Die Staged Configuration ist besonders sinnvoll, wenn man eine Variante als Grundlage für die weitere Arbeit nehmen möchte, d.h. der Product Tree übernimmt die Konfiguration. Nützlich wären außerdem automatische Benachrichtigungen für den Nutzer, sobald eine seiner Änderungen den Variantenraum verändert und die Definition externer Constraints, die kein *MassBudget* sind.

Eine weitere Herausforderung und wohl die auch die größte für Ingenieure, die sich noch nicht mit den Konzepten aus der FOSD beschäftigt haben, ist dass die Nutzung dieses Plugins aktuell noch erfordert, dass der Nutzer das Feature-Modell im Hinterkopf hat. So müssen manche Beziehungen (OR, Optional, Cross-Tree-Constraint) explizit im Feature Tree definiert werden. Beim Entfernen einer Element Definition und dem Auswählen der Beziehungsänderung wird die *XOR*-Beziehung zwischen allen Features der Baumebene erstellt, auf der die ED war. Es kann aber sein, dass nicht jedes dieser Features den selben Zweck wie das entfernte erfüllt und damit keine Alternative sein kann. Im AOCS z.B. sind GPSAntenna und GPSReceiver auf einer Ebene, können sich aber keinesfalls gegenseitig ersetzen. Damit die *XOR*-Beziehung korrekt funktioniert, muss der Nutzer also Alternativbauteile im Product Tree schon in eine eigene Baumebene verpacken, so wie es im obigen Beispiel mit *Control* gemacht wurde.

All das sind aber Funktionalitäten, die verbessert oder implementiert werden können. Ideen für die Umsetzung folgen im Kapitel 6.1. Die Grundfunktionalität der Verfolgung und Auswertung der Varianten des Entwurfsprozesses wurde zweifelsfrei belegt.

5.3 Re: Forschungsfragen

1. Wie lassen sich FOSD-Konzepte des Problem Space in den MBSE-Prozess von Virtual Satellite integrieren, sodass der Nutzer korrekte Varianten vorgeschlagen bekommt?

   Es lässt sich eine Schnittstelle in Virtual Satellite implementieren, die Informationen des Product Trees und Informationen aus Veränderungen des Product Trees in eine Form bringt, die in ein Feature-Modell übersetzt werden kann. Sobald das Feature-Modell auf valide Varianten überprüft ist, nimmt die Schnittstelle die Rückübersetzung in die Struktur aus Virtual Satellite als Configuration Tree vor.

2. Wie kann eine dynamische Produktlinie den Entwurfsprozess samt möglicher Varianten verfolgen?

   (a) Wie sieht das zugrundeliegende Feature-Modell aus?

   Das Feature-Modell hat die selbe Struktur wie der Product Tree, der sich an der Dekomposition eines Satelliten orientiert. Damit besteht es aus verschiedenen partiellen Konfigurationen, die ein Subsystem oder eine Funktionseinheit eines Satelliten darstellen. Das Feature-Modell übernimmt die Komponenten und hierarchischen Beziehungen aus dem Product Tree und die erweiterten Beziehungen des Feature Trees und übersetzt sie in logisch auswertbare Beziehungen zwischen den Features.

(b) An welcher Stelle wird das Feature-Modell definiert?

Das Feature-Modell wird definiert, sobald der Nutzer auf "Generate Variant Wizard" klickt. Dann werden die Beziehungen und Features des Feature Trees in logische Operatoren übersetzt, die als Feature-Modell abgebildet werden können.

(c) Wie kann das Feature Model dynamisch angepasst werden?

Der Feature Tree kann als eine anpassbare Abstraktionsebene des Feature-Modells betrachtet werden. Er ist die Schnittstelle zwischen dem dynamischen Entwurfsprozess, der durch viele kurzfristige Änderungen charakterisiert ist und dem Feature-Modell, das einmal erstellt wird und die Produktlinie definiert. Das Feature-Modell wird also jedes Mal neu erstellt, sobald "Generate Variant Wizard" geklickt wird. Da die Übersetzung in das Feature-Modell aber in Echtzeit (<1s) geschieht und die Struktur des Feature-Modells ähnlich bleibt, kann es als eine dynamische Anpassung des Feature-Modells betrachtet werden.

3. Wie sehen die Features aus?

(a) Welche Elemente des Raumfahrzeugentwurfs repräsentieren sie?

Die Features repräsentieren Funktionseinheiten eines Satelliten, wie sie auch im Product Tree definiert werden. Dabei kann es sich um ganze Subsysteme, funktionale Bauteile oder Software handeln.

(b) An welcher Stelle des Entwurfs werden sie definiert?

Die grundlegende Funktionseinheit wird als Element Definition oder Product Tree Domain im Product Tree definiert. Beim Generieren des Feature Trees erben sie ihre Informationen von den EDs/PTDs und werden als Feature instanziiert.

4. Wie können die Anforderungen und Constraints des Nutzers die Produktlinie beeinflussen?

Die Constraints können aktuell als Budget für den gesamten Feature Tree definiert werden. Die generierten Konfigurationen werden dann auf die Verletzung dieses Constraints überprüft und ggf. invalidiert.

5. Wie läuft der Prozess der Variantenselektion konzeptionell, im Code und für den Nutzer ab?

Konzeptionell wird das Feature-Modell auf alle validen Konfigurationen überprüft, eine Variante ausgewählt und dann als Configuration Tree instanziiert. Im Code werden die Beziehungen und Features des Feature Trees in die konjunktive Normalform im DIMACS-Format gebracht und anschließend durch einen SAT-Solver auf Lösungsmodelle untersucht. Die validen Konfiguration werden gespeichert, in Configuration Trees gecastet und nach Selektion durch den Nutzer zum Repository hinzugefügt. Der Nutzer klickt lediglich auf "Generate Variant Wizard", wählt im Wizard die Varian-

te aus, mit der er weiterarbeiten möchte und sie erscheint als Configuration Tree im Repository.

6. Wie wird mit Anforderungen umgegangen (d.h. es existiert keine mögliche Variante), die nicht erfüllt werden können?

   Handelt es sich um ein *void* Feature-Modell, erscheinen keine validen Varianten im Wizard. Das Fenster ist leer und es kann nichts ausgewählt werden.

6 Zusammenfassung

Das Ziel der Abschlussarbeit war die Erprobung einer Anwendung von Konzepten aus der Feature-orientierten Softwareentwcklung für den Raumfahrzeugentwurf, mit denen alle validen Varianten des Prozesses als Produkt einer Produktlinie dokumentiert werden. So soll das Abhandenkommen alter Varianten während des aktuellen iterativen Prozesses aus Virtual Satellite verhindert werden. Der Arbeitsablauf mit dem aktivierten FOSD-Plugin sollte dabei dem gewohnten Ablauf aus Virtual Satellite ähneln und komplett modular als eigenes *Concept* implementiert werden. Die Features und hierarchischen Beziehungen sollten aus dem Product Tree abgeleitet und die validen Varianten der Produktlinie als Configuration Trees instanziiert werden können. Der Entwurfsprozess sollte durch das Entfernen und Hinzufügen von Element Definitions und Product Tree Domains die Produktlinie, die als Feature-Modell dargestellt wird, mit weiteren Beziehungen zwischen den Features bereichern.

Das Ergebnis ist ein Feature Tree, der genau diese Schnittstelle in Virtual Satellite abbildet. Er erbt vom Product Tree und auch seine Features erben ihre Informationen und Position im Feature Tree von der zugehörigen ED oder PTD. Werden EDs oder PTDs im Product Tree gelöscht, entstehen automatisiert SubFeatureRelationships. Weitere Beziehungen zwischen den Features können direkt im Feature Tree definiert werden. Das Plugin übersetzt den Feature Tree in ein Feature-Modell, das in konjunktiver Normalform durch einen SAT-Solver auf alle validen Varianten untersucht wird. Diese Varianten werden dem Nutzer angezeigt und er kann sie als Configuration Tree instanziieren.

Dieses Konzept wurde anhand eines Beispiels erprobt. Das Konzept funktionierte als Virtual Satellite *Concept* und alle validen Varianten wurden gefunden. Während des Entwurfsprozesses gehen keine Varianten mehr verloren, sondern werden nun Teil der Produktlinie. Damit ist ein späterer Zugriff, eine Verschiebung von Entscheidungen und eine Observation des Variantenraums möglich.

Weitere Arbeit an der Automatisierung ist nötig, damit keine Beziehungen mehr direkt im Feature Tree definiert werden müssen und der Nutzer die Konzepte der Feature-orientierten Softwareentwicklung nicht mehr im Hinterkopf haben muss. Der Nutzer sollte außerdem automatisch auf die Konsequenzen seiner Veränderungen des Entwurfs aufmerksam gemacht werden, sodass er jederzeit einen Überblick über den Variantenraum hat.

6.1 Ausblick

Die vielversprechenden Ergebnisse können eine Zukunft für die Nutzung von FOSD-Konzepten in der Raumfahrzeugentwicklung ankündigen. Dafür muss vor allem die Automatisierung verbessert werden. Die *OR*-Beziehung und die OptionalRelationship können beide Teil des

Wizards werden, der den Nutzer fragt, ob die Beziehung des Features zu *XOR* geändert oder das Feature endgültig gelöscht werden soll. Cross-Tree-Constraints sollten mittels UI-Elementen wie gleicher Farbe der Knoten im Feature definiert werden können, damit der Ingenieur sich nicht um die UUID des Objekts kümmern muss. Die UI muss überarbeitet werden, sodass alle wichtigen Informationen wie gelöschte EDs oder valide Varianten angezeigt werden. Staged Configuration wäre mit einer Neuinstanziierung des Product Trees möglich. Die externen Constraints sollten mittels des *Requirements Specification Concepts* definiert werden können.

Literatur

[1] European Space Agency. *Model-based system engineering*. https://www.esa.int/Enabling_Support/Preparing_for_the_Future/Discovery_and_Preparation/Model-based_system_engineering. accessed: 10.03.2024. 2022.

[2] Philipp M. Fischer et al. *VirtualSatellite4-Core*. https://github.com/virtualsatellite/VirtualSatellite4-Core. 15.04.2024. 2023.

[3] Thomas Thüm et al. https://featureide.github.io.

[4] Sven Apel und Christian Kästner. "An Overview of Feature-Oriented Software Development". In: *Journal Of Object Technology* (2008).

[5] David Benavides, Serio Segura und Antonio Ruiz-Cortés. *Automated Analysis of Feature Models 20 Years Later: A Literature Review*. Techn. Ber. University of Seville, 2010.

[6] The Institute of Electrical und Electronics Engineers. *IEEE Guide to Software Requirements Specifications*. Techn. Ber. IEEE, 1984.

[7] ESA. *Open Concurrent Design Tool*. https://ocdt.esa.int. 17.03.2024. 2021.

[8] P.M. Fischer u. a. *Implementing Model Based Systems Engineering For The Whole Lifecycle Of A Spacecraft*. Techn. Ber. Deutsches Zentrum für Luft- und Raumfahrt, 2017.

[9] Peter W Fortescue, John Stark und Graham Swinerd. *Spacecraft Systems Engineering*. N.J: Wiley, 2011.

[10] Martin Holland. *Starlink Co.: Anzahl der Satelliten im Erdorbit steigt immer schneller*. https://www.heise.de/news/Starlink-Co-2022-mehr-Satelliten-ins-All-gestartet-als-je-zuvor-7448149.html. 22.03.2024. 2023.

[11] Deutsches Zentrum für Luft- und Raumfahrt. *VirSat$_C$ore$_U$ser$_M$anual*. https://github.com/virtualsatellite/VirtualSatellite4-Core/blob/development/de.dlr.sc.virsat.docs.feature/src/docs/VirSat_Core_User_Manual.adoc. 10.03.2024. 2023.

[12] Thomas von der Maßen und Horst Lichter. *Determining the Variation Degree of Feature Models*. Techn. Ber. RWTH Aachen, 2005.

[13] Klaus Pohl, Günter Böckle und Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[14] Ana Luísa Ramos, José Vasconcelos Ferreira und Jaume Barceló. *Model-Based Systems Engineering: An Emerging Approach for Modern Systems*. Techn. Ber. IEEE, 2012.

[15] Florida State University. *CNF Files*. https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html. 12.04.2024. 2008.

[16] Fabian Weilbrenner. *AUTOMATISIERTE ABLEITUNG VON VARIANTEN AUS SYSML-BASIERTEN SYSTEMARCHITEKTUREN*. 2016.

[17] Wikipedia. *Feature model - Semantics in Conjunctive normal form*. https://en.wikipedia.org/wiki/Feature_model. 05.03.2024. 2022.

A Anhang

A.1 FeatureIDE-Ergebnisse

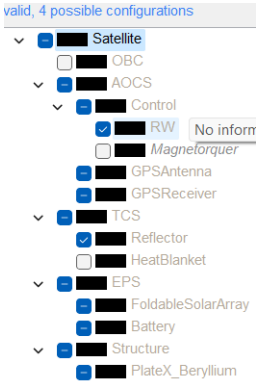
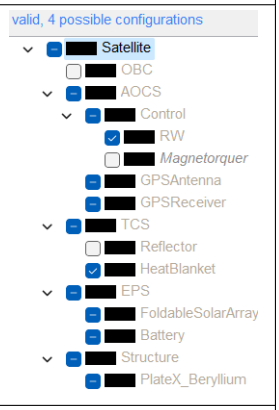
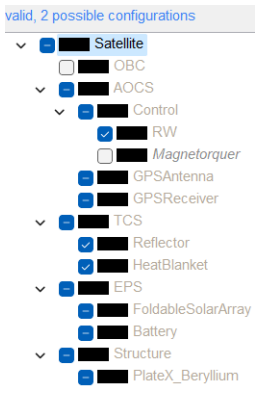
Variante	Feature-IDE
EC: AOCS EC: Control EC: RW EC: GPSAntenna EC: GPSReceiver EC: TCS EC: Reflector EC: EPS EC: FoldableSolarArray EC: Battery EC: Structure EC: PlateX_Beryllium	
EC: AOCS EC: GPSReceiver EC: Control EC: RW EC: GPSAntenna EC: TCS EC: HeatBlanket EC: EPS EC: FoldableSolarArray EC: Battery EC: Structure EC: PlateX_Beryllium	
EC: EPS EC: FoldableSolarArray EC: Battery EC: TCS EC: HeatBlanket EC: Reflector EC: AOCS EC: GPSReceiver EC: Control EC: RW EC: GPSAntenna EC: Structure EC: PlateX_Beryllium	

Tabelle 13: Ergebnisse aus der FeatureIDE (1/3)

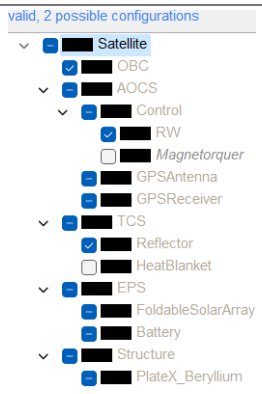
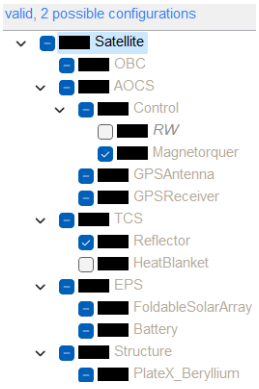
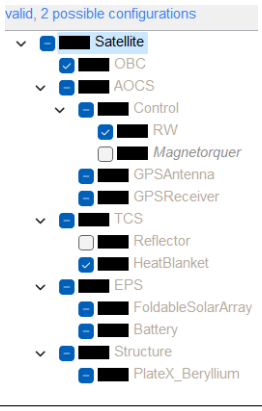
Variante	FeatureIDE
EC: OBC EC: EPS EC: Battery EC: FoldableSolarArray EC: AOCS EC: GPSReceiver EC: Control EC: RW EC: GPSAntenna EC: Structure EC: PlateX_Beryllium EC: TCS EC: Reflector	
EC: AOCS EC: Control EC: Magnetorquer EC: GPSReceiver EC: GPSAntenna EC: TCS EC: Reflector EC: Structure EC: PlateX_Beryllium EC: OBC EC: EPS EC: Battery EC: FoldableSolarArray	
EC: EPS EC: FoldableSolarArray EC: Battery EC: OBC EC: AOCS EC: GPSReceiver EC: Control EC: RW EC: GPSAntenna EC: TCS EC: HeatBlanket EC: Structure EC: PlateX_Beryllium	

Tabelle 14: Ergebnisse aus der FeatureIDE (2/3)

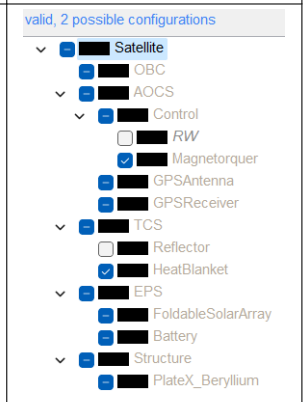
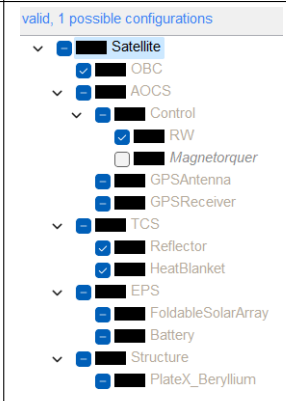
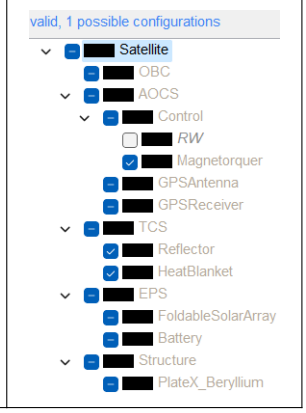
Variante	FeatureIDE
EC: TCS EC: HeatBlanket EC: OBC EC: Structure EC: PlateX_Beryllium EC: EPS EC: Battery EC: FoldableSolarArray EC: AOCS EC: GPSReceiver EC: GPSAntenna EC: Control EC: Magnetorquer	
EC: AOCS EC: GPSReceiver EC: GPSAntenna EC: Control EC: RW EC: OBC EC: EPS EC: FoldableSolarArray EC: Battery EC: Structure EC: PlateX_Beryllium EC: TCS EC: HeatBlanket EC: Reflector	
EC: Structure EC: PlateX_Beryllium EC: OBC EC: TCS EC: Reflector EC: HeatBlanket EC: EPS EC: FoldableSolarArray EC: Battery EC: AOCS EC: Control EC: Magnetorquer EC: GPSReceiver EC: GPSAntenna	

Tabelle 15: Ergebnisse aus der FeatureIDE (3/3)

A.2 VariantSelectionPage.java

```
1  public class VariantSelectionPage extends WizardPage {
2
3          private ISelection selection;
4          private ISelection preSelect;
5          private VirSatTransactionalEditingDomain ed;
6          private Repository rep;
7          private Composite content;
8          private StructuralElementInstance sc;
9          private StructuralElementInstance sei;
10         Map<Integer, StructuralElementInstance> variableToFeature = new HashMap<>();
11         Map<StructuralElementInstance, Integer> featureToVariable = new HashMap<>();
12         private List<int[]> validConfigurations;
13         private StructuralElementInstance selectedConfiguration;
14         private boolean generateAsConfigurationTree;
15         private Text treeName;
16         private List<Tree> tree = new ArrayList<>();
17         private TreeEditor editor;
18         /**
```

```
19          * Create a new Generate page
20          * @param preSelect the initial selection to be used as a model
21          */
22
23         protected VariantSelectionPage(ISelection preSelect) {
24
25                 super("");
26                 if (preSelect != null) {
27                         sc = (StructuralElementInstance) ((IStructuredSelection) preSelect).getFirstElement();
28                         if (sc.getType().getName().equals(FeatureTree.class.getSimpleName())) {
29                                 setTitle("Generate the Configuration Tree");
30                                 generateAsConfigurationTree = true;
31                         }
32                         setDescription("Right click on an element to rename or duplicate");
33                         this.preSelect = preSelect;
34                 }
35
36         }
37
38         @Override
39         public void createControl(Composite parent) {
40
41                 /*
42                  * If ConfigTree should be generate, make the variant selection and set the selected
43                  * variant as the new preselection.
44                  */
45
46                 if (generateAsConfigurationTree) {
47                         this.getAllValidVariants(sc);
48                 }
49
50                 content = new Composite(parent, SWT.NONE);
51                 GridLayout glContent = new GridLayout();
52                 glContent.numColumns = 2;
53                 content.setLayout(glContent);
54                 content.setLayoutData(new GridData(GridData.FILL_BOTH | GridData.GRAB_HORIZONTAL | GridData.
                        GRAB_VERTICAL));
55
56                 createTreeUI();
57             setControl(content);
58
59             ComposedAdapterFactory adapterFactory = new ComposedAdapterFactory(ComposedAdapterFactory.Descriptor.
                    Registry.INSTANCE);
60             adapterFactory.addAdapterFactory(new ResourceItemProviderAdapterFactory());
61                 adapterFactory.addAdapterFactory(new DVLMItemProviderAdapterFactory());
62                 adapterFactory.addAdapterFactory(new DVLMStructuralItemProviderAdapterFactory());
63                 adapterFactory.addAdapterFactory(new GeneralItemProviderAdapterFactory());
64                 adapterFactory.addAdapterFactory(new ConceptsItemProviderAdapterFactory());
65                 adapterFactory.addAdapterFactory(new RolesItemProviderAdapterFactory());
66                 adapterFactory.addAdapterFactory(new UnitsItemProviderAdapterFactory());
67                 adapterFactory.addAdapterFactory(new DVLMCategoriesItemProviderAdapterFactory());
68                 adapterFactory.addAdapterFactory(new PropertydefinitionsItemProviderAdapterFactory());
69                 adapterFactory.addAdapterFactory(new PropertyinstancesItemProviderAdapterFactory());
70                 adapterFactory.addAdapterFactory(new ReflectiveItemProviderAdapterFactory());
71
72             Composite composite = new Composite(content, SWT.NONE);
73             composite.setLayout(new GridLayout(2, false));
74             Button btnSelect = new Button(composite, SWT.NONE);
75             btnSelect.addSelectionListener(new SelectionAdapter() {
76                 @Override
77                 public void widgetSelected(SelectionEvent e) {
78                         StructuralElementInstance select = ((StructuralElementInstance) ((IStructuredSelection) selection).
                                getFirstElement()).getParent();
79
80                         selectedConfiguration = select;
81                         sei = selectedConfiguration;
82                 }
83             });
84             btnSelect.setText("Select");
85
86             new Label(content, SWT.NONE);
87             Label lblTreeName = new Label(content, SWT.NONE);
88             lblTreeName.setLayoutData(new GridData(SWT.RIGHT, SWT.CENTER, false, false, 1, 1));
89             treeName = new Text(content, SWT.BORDER);
90             treeName.setLayoutData(new GridData(SWT.FILL, SWT.CENTER, true, false, 1, 1));
91
92
93             if (generateAsConfigurationTree) {
94                 lblTreeName.setText("Configuration Tree Name");
95                 treeName.setText(ConfigurationTree.class.getSimpleName());
96                 }
97
98         }
99
100        /**
101         * Create the dialog for renaming and duplicating items
102         */
103        private void createTreeUI() {
104
```

```
105                 VirSatComposedContentProvider cp = new VirSatComposedContentProvider();
106                 cp.registerSubContentProvider(new VirSatWorkspaceContentProvider());
107                 cp.registerSubContentProvider(new VirSatProjectContentProvider());
108
109                 VirSatComposedLabelProvider lp = new VirSatComposedLabelProvider();
110                 lp.registerSubLabelProvider(new VirSatWorkspaceLabelProvider());
111                 lp.registerSubLabelProvider(new VirSatProjectLabelProvider());
112
113                 VirSatFilteredWrappedTreeContentProvider filteredCP = new VirSatFilteredWrappedTreeContentProvider(cp);
114                 filteredCP.setCheckContainedForFilter(true);
115
116                 // Filter for elements that will be Generated
117
118                 filteredCP.addClassFilter(StructuralElementInstance.class);
119                 filteredCP.addClassFilter(Repository.class);
120                 filteredCP.addClassFilter(VirSatProjectResource.class);
121
122                 // create the new Structural element
123                 StructuralElementInstance sc = (StructuralElementInstance) ((IStructuredSelection) preSelect).getFirstElement();
124                 this.ed = VirSatEditingDomainRegistry.INSTANCE.getEd(sc);
125                 VirSatResourceSet virSatResourceSet = ed.getResourceSet();
126                 this.rep = virSatResourceSet.getRepository();
127
128                 /*
129                  * Build the tree models for the view based on the valid configurations.
130                  */
131                 List<StructuralElementInstance> treeModels = new ArrayList<>();
132
133                 for (int[] config : validConfigurations) {
134                         // use of a mapper, so that we can access the newly added feature and get the related original feature.
135                         Map<StructuralElementInstance, StructuralElementInstance> featureToElementConfigurationMapper =
                                    new HashMap<>();
136
137                         ActiveConceptHelper acHelper = new ActiveConceptHelper(rep);
138                         Concept activeConcept = acHelper.getConcept(ProductStructureHelper.getConcept());
139
140                         // Create a configurationTree instance with repository as parent.
141                         ConfigurationTree ct = (ConfigurationTree) ProductStructureHelper.createTreeModel(sc);
142                         // We merely want to add the features of the specific configuration.
143                         ct.removeAllStructuralElementInstance(ct.getDeepChildren(IBeanStructuralElementInstance.class));
144                         StructuralElementInstance rootFeature = ct.getStructuralElementInstance();
145                         featureToElementConfigurationMapper.put(sc, rootFeature);
146
147                         // iterate through the features of the configuration.
148                         for (int i = 1; i < config.length; i++) {
149                                 if (config[i] > 0) {
150
151                                         // get actual feature of the FeatureTree
152                                         StructuralElementInstance actualFeature = variableToFeature.get(config[i]);
153                                         // find out parent
154                                         int actualParentVariable = featureToVariable.get(actualFeature.getParent());
155
156
157                                         // first feature has to be added to the root feature.
158                                         if (actualParentVariable == 1) {
159                                                 StructuralElementInstance feature = ProductStructureHelper.
                                                            createStructuralElementInstance(activeConcept, ElementConfiguration.
                                                            FULL_QUALIFIED_STRUCTURAL_ELEMENT_NAME, rootFeature);
160                                                 feature.setName(actualFeature.getName());
161                                                 feature.setType(acHelper.getStructuralElement(ElementConfiguration.
                                                            FULL_QUALIFIED_STRUCTURAL_ELEMENT_NAME));
162                                                 rootFeature.getChildren().add(feature);
163
164                                                 featureToElementConfigurationMapper.put(actualFeature, feature);
165                                         } else {
166                                                 StructuralElementInstance parentFeature = featureToElementConfigurationMapper
                                                            .get(variableToFeature.get(actualParentVariable));
167
168                                                 StructuralElementInstance feature = ProductStructureHelper.
                                                            createStructuralElementInstance(activeConcept, ElementConfiguration.
                                                            FULL_QUALIFIED_STRUCTURAL_ELEMENT_NAME, parentFeature);
169                                                 feature.setName(actualFeature.getName());
170                                                 feature.setType(acHelper.getStructuralElement(ElementConfiguration.
                                                            FULL_QUALIFIED_STRUCTURAL_ELEMENT_NAME));
171                                                 parentFeature.getChildren().add(feature);
172
173                                                 featureToElementConfigurationMapper.put(actualFeature, feature);
174                                         }
175
176                                 }
177                         }
178                         treeModels.add(rootFeature);
179                 }
180
181                 // Create the tree models for the configurations.
182                 for (StructuralElementInstance treeModel : treeModels) {
183                         TreeViewer treeViewer = new TreeViewer(content, SWT.BORDER);
184                         treeViewer.setComparator(new VirSatNavigatorSeiSorter());
185                         treeViewer.setContentProvider(filteredCP);
```

```
186                         treeViewer.setLabelProvider(lp);
187                         treeViewer.setInput(treeModel);
188                         treeViewer.expandAll();
189                         tree.add(treeViewer.getTree());
190                         editor = new TreeEditor(treeViewer.getTree());
191
192                         treeViewer.addSelectionChangedListener(new ISelectionChangedListener() {
193                                 @Override
194                                 public void selectionChanged(SelectionChangedEvent event) {
195                                         Control oldEditor = editor.getEditor();
196                                         if (oldEditor != null) {
197                                                 oldEditor.dispose();
198                                         }
199                                         selection = event.getSelection();
200
201                                 }
202                         });
203                 }
204         }
205
206         /*
207          * Takes a FeatureTree instance, translates it to conjunctive normal form and feeds it to a SAT solver
208          * to retrieve valid variants of the feature model.
209          * @param StructuralElementInstance the FeatureTree to be evaluated
210          */
211         public void getAllValidVariants(StructuralElementInstance featureTree) {
212                 /*
213                  * Translate to cnf
214
215                         r is root feature | r
216                         p is parent of feature c | c -> p
217                         m is mandatory subfeature of p | p -> m
218                         p is the parent of [1..n] grouped features g1, ... , gn | p -> (g1 V ... V gn)
219                         p is parent of [1..1] grouped features g1, ... , gn | p -> 1-of-n(g1, ... , gn)
220                         Cross-tree constraint c requires p | c -> p
221                         Cross-tree constraint c excludes p | c -> -p
222                 */
223
224                 /*
225                  * Map features to variables of the CNF.
226                  * Root Feature = 1 and so on.
227                  */
228
229                 // put root feature
230                 variableToFeature.put(1, featureTree);
231                 featureToVariable.put(featureTree, 1);
232
233                 // next variable is number 2
234                 int variableCounter = 2;
235
236                 for (StructuralElementInstance feature : featureTree.getDeepChildren()) {
237                         variableToFeature.put(variableCounter, feature);
238                         featureToVariable.put(feature, variableCounter);
239                         variableCounter++;
240                 }
241
242                 /*
243                  * Use ModelIterator to find all valid variants.
244                  */
245                 ISolver solver = new ModelIterator(SolverFactory.newDefault());
246
247                 // List for preventing duplicated SubFeatureRelationship clauses
248                 List<Integer> existingSubFeatureRelationships = new ArrayList<Integer>();
249
250                 /*
251                  * Iterate through the features and view every feature as child node
252                  * (thus, see parent node for SubFeatureRelationship).
253                  */
254                 for (Entry<Integer, StructuralElementInstance> feature : variableToFeature.entrySet()) {
255
256                         // root is not a child
257                         if (feature.getKey() == 1) {
258                                 continue;
259                         }
260
261
262                         /*
263                          * CategoryAssignments of feature.
264                          */
265                         Optional<CategoryAssignment> optionalRelationship = feature.getValue().getCategoryAssignments().
                                stream()
266                                         .filter(ca -> ca.getType().getName().equals("OptionalRelationship"))
267                                         .findAny();
268
269                         Optional<CategoryAssignment> crossTreeConstraint = feature.getValue().getCategoryAssignments().
                                stream()
270                                         .filter(ca -> ca.getType().getName().equals("CrossTreeConstraint"))
271                                         .findAny();
272
```

```
273                     /*
274                      * SubFeatureRelationship of parent.
275                      */
276                     StructuralElementInstance parent = feature.getValue().getParent();
277                     Optional<CategoryAssignment> parentSubFeatureRelationship = parent.getCategoryAssignments().stream
                            ()
278                                     . filter (ca -> ca.getType().getName().equals("SubFeatureRelationship"))
279                                     .findAny();
280
281                     /*
282                      * Handle SubFeatureRelationship.
283                      */
284                     if (parentSubFeatureRelationship.isPresent()) {
285
286                             SubFeatureRelationship subFeatureRelationship = new SubFeatureRelationship(
                                    parentSubFeatureRelationship.get());
287
288                             if (subFeatureRelationship.getCharacter() != null && !existingSubFeatureRelationships.contains(
                                    feature.getKey())) {
289                                     // For SubFeatureRelationships we need the other affected feature variables.
290                                     int[] subFeatureVariables = parent.getChildren().stream().mapToInt(child ->
                                            featureToVariable.get(child)).toArray();
291
292                                     // Add parent variable to express the relation to parent feature.
293                                     int[] subFeaturePlusParent = new int[subFeatureVariables.length+1];
294                                     for (int i = 0; i < subFeatureVariables.length; i++) {
295                                             subFeaturePlusParent[i] = subFeatureVariables[i];
296                                     }
297                                     subFeaturePlusParent[subFeaturePlusParent.length-1] = featureToVariable.get(parent);
298
299                                     switch(subFeatureRelationship.getCharacter()) {
300                                             case "xor" -> handleXor(subFeaturePlusParent, solver);
301                                             case "or" -> handleOr(subFeaturePlusParent, solver);
302                                     }
303
304                                     existingSubFeatureRelationships.addAll(IntStream.of(subFeatureVariables).boxed().collect(
                                            Collectors.toList()));
305                             }
306
307                     /*
308                      * Create new clauses when the parent has no SubFeatureRelationship which affects this feature.
309                      */
310                     } else {
311
312                             // If value of PropertyInstance is set to true
313                             boolean optionalFlagSet = true;
314
315                             /*
316                              * Handle Optional.
317                              */
318                             if (optionalRelationship.isPresent()) {
319
320                                     OptionalRelationship optionalInstance = new OptionalRelationship(optionalRelationship.
                                            get());
321
322                                     // Check if value is "true", else the relationship is mandatory
323                                     if (optionalInstance.getIsOptional()) {
324                                             handleOptional(featureToVariable.get(parent), feature.getKey(), solver);
325                                     }
326                                     else {
327                                             optionalFlagSet = false;
328                                     }
329
330                             /*
331                              * If either no OptionalRelationship is defined or it is set to false, we assume the relationship
                                     is mandatory.
332                              */
333                             } else if (optionalRelationship.isEmpty() || !optionalFlagSet) {
334                                     handleMandatory(featureToVariable.get(parent), feature.getKey(), solver);
335                             }
336
337                     }
338
339                     /*
340                      * Handle CrossTreeConstraint.
341                      */
342                     if (crossTreeConstraint.isPresent()) {
343
344                             CrossTreeConstraint crossTreeConstraintInstance = new CrossTreeConstraint(crossTreeConstraint.
                                    get());
345
346                             if (crossTreeConstraintInstance.getReferenceUuid() != null) {
347
348                                     // Get other feature variable by uuid
349                                     int referencedFeatureVariable = 0;
350                                     for (Entry<Integer, StructuralElementInstance> otherFeature : variableToFeature.
                                            entrySet()) {
351                                             if (otherFeature.getValue().getUuid().toString().equals(
                                                    crossTreeConstraintInstance.getReferenceUuid())) {
```

```
352                                                     referencedFeatureVariable = otherFeature.getKey();
353                                             }
354                                     }
355
356                                     switch (crossTreeConstraintInstance.getCharacterBean().getValue()) {
357                                             // Requires
358                                             case "enumValue1" -> handleRequires(referencedFeatureVariable, feature.getKey()
                                                    , solver);
359                                             // Excludes
360                                             case "enumValue2" -> handleExcludes(referencedFeatureVariable, feature.getKey()
                                                    , solver);
361                                     }
362                             }
363                     }
364             }
365
366
367             /*
368              * Solve until all solutions are found
369              */
370              boolean unsat = true;
371              List<int[]> models = new ArrayList<>();
372              try {
373                     while (solver.isSatisfiable()) {
374                      unsat = false;
375                      models.add(solver.model());
376                      // do something with model
377                      }
378             } catch (TimeoutException e) {
379                     e.printStackTrace();
380             }
381              if (unsat) {
382                  // UNSAT case
383              }
384
385              /*
386               * Remove first entry of list, because it is case of every feature being negated.
387               */
388              models.remove(0);
389
390              /*
391               * Check if variant exceeds mass budget.
392               */
393              Optional<CategoryAssignment> massBudgetOpt = featureTree.getCategoryAssignments().stream()
394                                      .filter(ca -> ca.getType().getName().equals("MassBudget"))
395                                      .findAny();
396              if (massBudgetOpt.isPresent()) {
397                     MassBudget massBudget = new MassBudget(massBudgetOpt.get());
398                     List<int[]> modelsToRemove = new ArrayList<>();
399
400                     for (int[] model : models) {
401                             double cumMass = 0.0;
402
403                             for (int i = 0; i < model.length; i++) {
404                                     if (model[i] > 0) {
405                                             Optional<CategoryAssignment> mass = variableToFeature.get(model[i]).
                                                    getCategoryAssignments().stream()
406                                                             .filter(ca -> ca.getType().getName().equals("MassEquipment"))
407                                                             .findAny();
408
409                                             if (mass.isPresent()) {
410                                                     MassEquipment massEquipment = new MassEquipment(mass.get());
411                                                     cumMass += massEquipment.getMass();
412                                             }
413
414                                     }
415                                     if (cumMass > massBudget.getMass()) {
416                                             modelsToRemove.add(model);
417                                     }
418                             }
419
420                     }
421                     models.removeAll(modelsToRemove);
422              }
423              validConfigurations = models;
424     }
425
426     /*
427      * Handle mandatory relationship.
428      * (-parent or child) and (-child or parent)
429      */
430     public void handleMandatory(int parentVariable, int childVariable, ISolver solver) {
431             try {
432                     solver.addClause(new VecInt(new int[]{-parentVariable, childVariable})).toString();
433                     solver.addClause(new VecInt(new int[]{-childVariable, parentVariable})).toString();
434                     System.out.println(-parentVariable + " " + childVariable);
435                     System.out.println(parentVariable + " " + -childVariable);
436             } catch (ContradictionException e) {
437                     e.printStackTrace();
```

```
438                 }
439         }
440
441         /*
442          * Handle optional relationship.
443          * (-child or parent)
444          */
445         public void handleOptional(int parentVariable, int childVariable, ISolver solver) {
446                 try {
447                         solver.addClause(new VecInt(new int[]{-childVariable, parentVariable}));
448                         System.out.println(parentVariable + " " + -childVariable);
449                 } catch (ContradictionException e) {
450                         e.printStackTrace();
451                 }
452         }
453
454         /*
455          * Handle requires cross tree constraint.
456          * (-optional or required)
457          */
458         public void handleRequires(int requiredVariable, int optionalVariable, ISolver solver) {
459                 handleOptional(requiredVariable, optionalVariable, solver);
460         }
461
462         /*
463          * Handle excludes cross tree constraint.
464          * (-y or -x)
465          */
466         public void handleExcludes(int excludedVariable, int thisVariable, ISolver solver) {
467                 try {
468                         solver.addClause(new VecInt(new int[]{-excludedVariable, -thisVariable}));
469                         System.out.println(-excludedVariable + " " + -thisVariable);
470                 } catch (ContradictionException e) {
471                         e.printStackTrace();
472                 }
473         }
474
475         /*
476          * Handle XOR relationship.
477          * (child_1 or ... child_n or -parent) and
478          * for i..n(-child_i or parent) and
479          * for i<j(-child_i or -child_j)
480          */
481         public void handleXor(int[] subFeatures, ISolver solver) {
482                 //solver.addParity(new VecInt(subFeatures), true).toString();
483                 int [] children = new int[subFeatures.length-1];
484
485                 int parent = subFeatures[subFeatures.length-1];
486
487                 for (int i = 0; i < children.length; i++) {
488                         children[i] = subFeatures[i];
489                 }
490
491                 /*
492                  * Clause containing all subFeatures, form:
493                  * (child_1 or ... child_n or -parent)
494                  */
495                 try {
496                         int [] copy = subFeatures.clone();
497                         copy[copy.length-1] = -copy[copy.length-1];
498                         IVecInt v = new VecInt(copy);
499                         solver.addClause(v);
500                         for (int i : copy) {
501                                 System.out.print(i + " ");
502                         }
503                         System.out.println();
504                 } catch (ContradictionException e1) {
505                         e1.printStackTrace();
506                 }
507
508                 /*
509                  * Individual clauses for each child, form:
510                  * (-child_i or parent)
511                  */
512
513                 for (int i = 0; i < children.length; i++) {
514                         try {
515                                 IVecInt v = new VecInt(new int[]{-children[i], parent});
516                                 solver.addClause(v);
517                                 System.out.println(-children[i] + " " + parent);
518                         } catch (ContradictionException e) {
519                                 e.printStackTrace();
520                         }
521                 }
522                 /*
523                  * Individual clauses to prevent children from being the same, form:
524                  * all_i<j(-child_i or -child_j)
525                  */
526                 for (int i = 0; i < children.length; i++) {
```

```
527                         for (int j = i+1; j < children.length; j++) {
528                                 try {
529                                         IVecInt v = new VecInt(new int[]{-children[i], -children[j]});
530                                         solver.addClause(v);
531                                         System.out.println(-children[i] + " " + children[j]);
532                                 } catch (ContradictionException e) {
533                                         e.printStackTrace();
534                                 }
535                         }
536                 }
537         }
538
539         /*
540          * Handle OR relationship.
541          * (child_1 or ... child_n or -parent) and
542          * for i..n(-child_i or parent)
543          */
544         public void handleOr(int[] subFeatures, ISolver solver) {
545
546                 int[] children = new int[subFeatures.length-1];
547                 int parent = subFeatures[subFeatures.length-1];
548
549                 for (int i = 0; i < children.length; i++) {
550                         children[i] = subFeatures[i];
551                 }
552
553                 /*
554                  * Clause containing all subFeatures, form:
555                  * (child_1 or ... child_n or -parent)
556                  */
557                 try {
558                         int[] copy = subFeatures.clone();
559                         copy[copy.length-1] = -copy[copy.length-1];
560                         IVecInt v = new VecInt(copy);
561                         solver.addClause(v);
562                         for (int i : copy) {
563                                 System.out.print(i + " ");
564                         }
565                         System.out.println();
566                 } catch (ContradictionException e1) {
567                         e1.printStackTrace();
568                 }
569
570                 /*
571                  * Individual clauses for each child, form:
572                  * (-child_i or parent)
573                  */
574                 for (int i = 0; i < children.length; i++) {
575                         try {
576                                 IVecInt v = new VecInt(new int[]{-children[i], parent});
577                                 solver.addClause(v);
578                                 System.out.println(-children[i] + " " + parent);
579                         } catch (ContradictionException e) {
580                                 e.printStackTrace();
581                         }
582                 }
583         }
```

A.3 UpdateFeatureTree.java

```
1   public class UpdateFeatureTreeHandler extends AbstractHandler implements IHandler{
2
3           @Override
4           public Object execute(ExecutionEvent event) throws ExecutionException {
5                   ISelection selection = HandlerUtil.getCurrentSelectionChecked(event);
6                   StructuralElementInstance sc = (StructuralElementInstance) ((IStructuredSelection) selection).getFirstElement();
7                   VirSatTransactionalEditingDomain ed = VirSatEditingDomainRegistry.INSTANCE.getEd(sc);
8                   VirSatResourceSet resSet = (VirSatResourceSet) ed.getResourceSet();
9                   Repository repository = resSet.getRepository();
10                  ActiveConceptHelper ach = new ActiveConceptHelper(repository);
11
12                  /*
13                   * Contains the product tree if present.
14                   */
15                  StructuralElementInstance productTree = null;
16
17                  /*
18                   * Contains the feature tree if present.
19                   */
20                  StructuralElementInstance featureTree = null;
21
22                  /*
23                   * Contains all elements of the product tree.
24                   */
25                  List<StructuralElementInstance> productTreeSEs = new ArrayList<StructuralElementInstance>();
26
```

```
27                  /*
28                   * Contains the elements removed from the product tree.
29                   */
30                  List<StructuralElementInstance> removedFromProductTree = new ArrayList<StructuralElementInstance>();
31
32                  for (StructuralElementInstance se : repository.getRootEntities()) {
33                          if (se.getType().getName().equals(ProductTree.class.getSimpleName())) {
34                                  productTree = se;
35                          }
36                          if (se.getType().getName().equals(FeatureTree.class.getSimpleName())) {
37                                  featureTree = se;
38                          }
39                  }
40
41                  productTreeSEs.addAll(productTree.getDeepChildren());
42
43                  /*
44                   *  Check for every feature whether the super ElementDefinition
45                   *  is  still  in the product tree.
46                   *  Removed EDs will later be prompted to decide whether
47                   *  terminally removing them or changing the relationship.
48                   */
49                  for (StructuralElementInstance child : featureTree.getDeepChildren()) {
50                          if (child.getSuperSeis().size() == 0) {
51                                  removedFromProductTree.add(child);
52                          }
53                  }
54
55
56
57                  // compare size of trees
58                  // if  different , ElementDefinitions were added
59                  if (featureTree.getDeepChildren().size() < (productTreeSEs.size() + removedFromProductTree.size())) {
60                          List<StructuralElementInstance> featuresToAdd = getAddedElementDefinitions(productTree.
                                    getDeepChildren(), featureTree.getDeepChildren());
61
62                          EList<StructuralElementInstance> nodesInFeatureTree = featureTree.getDeepChildren();
63                          nodesInFeatureTree.add(featureTree);
64
65                          /*
66                          while (!featuresToAdd.isEmpty()) {
67                                  for (StructuralElementInstance featureToAdd : featuresToAdd) {
68                                          StructuralElementInstance parent = findParentInFeatureTree(nodesInFeatureTree,
                                                    featureToAdd);
69                                          if (parent == null) {
70                                                  continue;
71                                          }
72                                          addFeatureToFeatureTree(parent, featureToAdd);
73                                          featuresToAdd.remove(featureToAdd);
74                                  }
75                          }*/
76
77
78                          while (!featuresToAdd.isEmpty()) {
79                                  Iterator<StructuralElementInstance> i = featuresToAdd.iterator();
80                                  while (i.hasNext()) {
81                                          StructuralElementInstance featureToAdd = i.next();
82                                          StructuralElementInstance parent = findParentInFeatureTree(nodesInFeatureTree,
                                                    featureToAdd);
83                                          if (parent == null) {
84                                                  continue;
85                                          }
86                                          addFeatureToFeatureTree(parent, featureToAdd);
87                                          i.remove();
88                                  }
89                          }
90                  }
91
92                  /*
93                   * Check if feature  model is void
94                   */
95
96                  if (!removedFromProductTree.isEmpty()) {
97                          for (StructuralElementInstance sei : removedFromProductTree) {
98                                  Optional<CategoryAssignment> subFeatureRelationship = sei.getParent().getCategoryAssignments
                                            ().stream()
99                                                  . filter (ca -> ca.getType().getName().equals("SubFeatureRelationship"))
100                                                 .findAny();
101                                 if (subFeatureRelationship.isEmpty()) {
102                                         Shell  shell  = HandlerUtil.getActiveWorkbenchWindow(event).getShell();
103                                         ChangeRelationOrDeleteWizard wizard = new ChangeRelationOrDeleteWizard();
104                                         wizard.setSei(sei);
105                                         WizardDialog dialog = new WizardDialog(shell, wizard);
106                                         dialog.open();
107                                 }
108                         }
109                 }
110                 /*
111                  * Check if feature  model is void
```

```
112                  */
113                 return null;
114
115         }
116
117         public void addFeatureToFeatureTree(StructuralElementInstance parent, StructuralElementInstance inheritance) {
118                 VirSatTransactionalEditingDomain ed = VirSatEditingDomainRegistry.INSTANCE.getEd(parent);
119                 Concept concept = ActiveConceptHelper.getConcept(parent.getType());
120
121
122                 // Create Feature and add inheritance
123                 String structuralElementName = "Feature";
124                 StructuralElement structuralElement = ActiveConceptHelper.getStructuralElement(concept,
                            structuralElementName);
125                 StructuralInstantiator structuralInstantiator = new StructuralInstantiator();
126                 StructuralElementInstance structuralElementInstance = structuralInstantiator.generateInstance(structuralElement,
                            structuralElementName);
127                 structuralElementInstance.setAssignedDiscipline(parent.getAssignedDiscipline());
128                 structuralElementInstance.getSuperSeis().add(inheritance);
129                 structuralElementInstance.setName(inheritance.getName());
130
131                 // Create command and execute
132                 Command addStructuralElementInstance = CreateAddSeiWithFileStructureCommand.create(ed, parent,
                            structuralElementInstance);
133                 ed.getCommandStack().execute(addStructuralElementInstance);
134
135                 // Try to open the CA in case it is preferred
136                 PreferencesEditorAutoOpen.openEditorIfPreferredForCollection(addStructuralElementInstance.getResult());
137         }
138
139         /*
140          * Get the difference between two children lists.
141          */
142         private List<StructuralElementInstance> getAddedElementDefinitions(EList<StructuralElementInstance>
                    productTreeDeepChildren,
143                         EList<StructuralElementInstance> featureTreeDeepChildren) {
144                 Set<StructuralElementInstance> ptSet = new HashSet<StructuralElementInstance>(productTreeDeepChildren);
145                 Set<StructuralElementInstance> ftSet = new HashSet<StructuralElementInstance>();
146
147                 // we need to compare with the superSEI, otherwise all elements are different
148                 for (StructuralElementInstance ftChild : featureTreeDeepChildren) {
149                         if (ftChild.getSuperSeis().size() != 0) {
150                                 ftSet.add(ftChild.getSuperSeis().get(0));
151                         }
152                 }
153
154                 ptSet.removeAll(ftSet);
155                 List<StructuralElementInstance> addedElementDefinitions = new ArrayList<StructuralElementInstance>(ptSet);
156                 return addedElementDefinitions;
157         }
158
159         private StructuralElementInstance findParentInFeatureTree(List<StructuralElementInstance> featureTreeNodes,
160                         StructuralElementInstance elementInProductTree) {
161                 // search for node in feature tree which superSEI is the parent of the elementInProductTree
162                 for (StructuralElementInstance node : featureTreeNodes) {
163                         EList<StructuralElementInstance> superSei = node.getSuperSeis();
164                         if (superSei.size() != 0) {
165                                 if (superSei.get(0).equals(elementInProductTree.getParent())) {
166                                         return node;
167                                 }
168                         }
169                 }
170                 return null;
171         }
172
173 }
```

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, 22.04.2024