Otto-von-Guericke Universität Magdeburg

Fakultät der Informatik

Masterarbeit

Konfiguration von Softwareproduktlinien mit Configuring-Constraints über Feature-Attributen

Autor:

Daniel Hohmann

2. Mai 2022

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake, M.Sc. Elias Kuiter
Institut für Technische und Betriebliche Informationssysteme
Otto-von-Guericke-Universität Magdeburg

Prof. Dr.-Ing. Thomas Thüm, M.Sc. Sebastian Krieter
Institut für Softwaretechnik und Programmiersprachen
Universität Ulm

Hohmann, Daniel:
Konfiguration von Softwareproduktlinien mit Configuring-Constraints über Feature-Attributen
Masterarbeit, Otto-von-Guericke Universität Magdeburg, 2022.

Zusammenfassung

Die Bedeutung von variablen Softwaresystemen ist in der Softwareindustrie stark gestiegen. Mittels Software-Produktlinien (SPLs) kann hier Abhilfe geschaffen werden. In SPLs werden die enthaltenen Features und ihre Abhängigkeiten (Constraints) in Feature-Modellen festgehalten. Dabei können zusätzliche Informationen wie zum Beispiel Preisinformationen zu den Features in Form von Feature-Attributen erfasst werden. Allerdings haben Nutzer beim Konfigurieren haben Nutzer bisher noch keine Möglichkeit, komplexe Constraints basierend auf diesen Feature-Attributen anzugeben.

Wir stellen in dieser Arbeit ein Konzept vor, mit dem Constraints über Feature-Attributen erstellt und in den Konfigurationsprozess eingebunden werden können. Wir stellen dabei Aggregationsfunktionen vor, die die Aggregation der Feature-Attribute der ausgewählten Features übernehmen. Für die Evaluierung setzen wir unser Konzept prototypisch um und ermitteln in verschiedenen Experimenten, ob sich unser Vorgehen auf die Größe der Formeln und die Zeit, die der SMT-Solver für das Finden einer Lösung benötigt, auswirkt. Wir stellen dabei fest, dass das Konzept für kleinere und mittlere Modelle tauglich, für größere Modelle jedoch nicht praktikabel ist.

Danksagungen

Mein Dank gilt all den Personen, die mich bei der Auseinandersetzung mit der Thematik und bei der Erstellung der vorliegenden Arbeit unterstützt und begleitet haben. An erster Stelle möchte ich mich bei Gunter Saake und Thomas Thüm bedanken, die mir die Möglichkeit zur Bearbeitung dieses Themas gegeben haben. Ein besonderer Dank geht an Elias Kuiter und Sebastian Krieter, die mir bei der Erstellung der Arbeit mit Hilfestellungen und den ein oder anderen kritischen Fragen beigestanden haben.

Inhaltsverzeichnis

Abbildungsverzeichnis ix

Tabellenverzeichnis xi

Quelltext xiii

1 Einführung 1

2 Grundlagen 5
- 2.1 Software-Produktlinien 5
- 2.2 Feature-orientierte Softwareentwicklung 6
- 2.3 Feature-Modellierung 8
  - 2.3.1 Feature-Modelle 8
  - 2.3.2 Feature-Attribute 9
  - 2.3.3 Constraints 11
- 2.4 Feature-Modell-Analyse 11

3 Konzept 15
- 3.1 Motivation 15
  - 3.1.1 Model-Attribute und Model-Constraints 16
  - 3.1.2 Configuration-Attribute und Model-Constraints 17
  - 3.1.3 Model-Attribute und Configuration-Constraints 17
  - 3.1.4 Configuration-Attribute und Configuration-Constraints 18
  - 3.1.5 Anwendungsfall der Arbeit 18
- 3.2 Formale Definition 19
  - 3.2.1 Syntax 19
  - 3.2.2 Feature-Attribute 20
  - 3.2.3 Aggregationsfunktionen 22
- 3.3 Anwendungsbeispiel 24
- 3.4 Abbildung auf SMT-Syntax 26

4 Implementierung 31
- 4.1 Umwandeln des Feature-Modells 31
  - 4.1.1 Erweitertes Feature-Modell 31
  - 4.1.2 Feature-Attribute 33
  - 4.1.3 Variablen einfügen 34
    - 4.1.3.1 Hilfsvariable für das arithmetische Mittel 34

4.1.3.2 Variablen für Feature-Attribute in Hinblick auf Aggregationsfunktionen . . . 34
4.1.4 Werte für Variablen einschränken . . . 35
4.2 Anwenden von Configuration-Constraints . . . 35
4.2.1 Format für Configuration-Constraints . . . 36
4.2.2 Umsetzen der Constraints mit JavaSMT . . . 38

5 Evaluierung **41**
5.1 Forschungsfragen . . . 41
5.2 Verwendete Feature-Modelle und Versuchsaufbau . . . 43
5.2.1 Modelle . . . 43
5.2.2 Durchführung . . . 44
5.2.3 Technische Voraussetzungen . . . 45
5.3 Ergebnisse . . . 45
5.4 Angriffspunkte . . . 56
5.4.1 Bedrohungen der internen Validität . . . 56
5.4.2 Bedrohungen der externen Validität . . . 57

6 Verwandte Arbeiten **59**

7 Fazit **61**

Anhang **63**

Literaturverzeichnis **65**

Abbildungsverzeichnis

2.1 Entwicklungsprozess einer SPL 7

2.2 Feature-Diagramm eines Webservers 8

2.3 Formalisierung des Beispiels aus Abbildung 2.2 12

3.1 Anwendungsfälle kategorisiert nach Constraints und Feature-Attributen 16

4.1 Umwandeln des Feature-Modells in eine aussagenlogische Formel . . . 32

5.1 Anzahl der Literale verschiedener Feature-Modelle in Bezug auf das jeweilige Modell ohne Attribute und Count (FF1) 46

5.2 Zeiten für die Umwandlung von Feature-Modellen und das Finden einer Lösung mit Z3 bezogen auf das jeweilige Feature-Modell ohne Attribute und Count 48

5.3 Zeiten für das Umwandeln und Finden von Lösungen für das Modell *webserver* mit Configuration-Constraints mit versch. Aggr.-Funktionen relativ zur Zeit ohne Configuration-Constraints 50

5.4 Zeiten für das Umwandeln und Finden von Lösungen für das Modell *sandwich* mit Configuration-Constraints mit versch. Aggr.-Funktionen relativ zur Zeit ohne Configuration-Constraints 51

5.5 Zeiten für das Umwandeln und Finden von Lösungen für das Modell *pc_config* mit Configuration-Constraints mit versch. Aggr.-Funktionen relativ zur Zeit ohne Configuration-Constraints 52

5.6 Zeiten für das Umwandeln und Finden von Lösungen für das Modell *webserver* mit Configuration-Constraint mit Kombinationen von Aggregationsfunktionen relativ zur Zeit ohne Configuration-Constraints 54

5.7 Zeiten für das Umwandeln und Finden von Lösungen für das Modell *sandwich* mit Configuration-Constraint mit Kombinationen von Aggregationsfunktionen relativ zur Zeit ohne Configuration-Constraints 55

5.8 Zeiten für das Umwandeln und Finden von Lösungen für das Modell *pc_config* mit Configuration-Constraint mit Kombinationen von Aggregationsfunktionen relativ zur Zeit ohne Configuration-Constraints 56

Tabellenverzeichnis

2.1 Beispielhafte Werte für die Attribute *Latenz*, *MAnf*, *Ausfall* 10

3.1 Konstanten für die Werte des Feature-Attributs *Latenz* basierend auf den Werten aus Tabelle 2.1 20

3.2 Standardwerte für Aggregationsfunktionen über numerischen Feature-Attributen 23

3.3 Variablennamen für das Feature *MySQL* für das Feature-Attribut *Ausfall* 30

4.1 Benötigte Logiken der SMTLib [Barrett et al., 2015] 39

5.1 Verwendete Modelle mit Feature- und Feature-Attribut-Anzahl 44

5.2 Verwendete Modelle nach Feature-, Literal-, Feature-Attribut und Variablen-Anzahl nach der Umwandlung 45

Quelltext

4.1 Wurzel-Feature aus Abbildung 2.2 als XML 32
4.2 Beispielhaftes Cross-Tree-Constraint als XML 32
4.3 Feature-Attribut *Latenz* vom Webserver als XML 33
4.4 Grundlegene XML-Struktur für Configuration-Constraints 36
4.5 Beispiel für Konstanten in Configuration-Constraints 36
4.6 Beispiel für Aggregationsfunktionen in Configuration-Constraints . . 37
4.7 Beispiel einer kompletten XML-Struktur für die SPL aus Abbildung 2.2 38
A.1 Feature-Modell aus Abbildung 2.2 im XML-Format von FeatureIDE . 63

1. Einführung

Die Bedeutung von variablen Softwaresystemen ist in der Softwareindustrie stark gestiegen [Fischer et al., 2014; Pohl et al., 2005]. Software-Produktlinien (SPLs) liefern die Möglichkeit, diese Systeme zu planen, zu erstellen und zu warten [Apel et al., 2013]. Die Features, die in den SPLs die Konfigurationsoptionen bereitstellen und damit die Variabilität der Softwaresysteme realisieren, werden in Feature-Modellen festgehalten. Die Beziehungen der Features untereinander werden mittels Constraints dargestellt.

Im Laufe der Zeit haben sich verschiedene textuelle Sprachen etabliert [Abele et al., 2010; Acher et al., 2013; Alférez et al., 2019; Azzawi und Fouad, 2018; Batory, 2005; Clarke et al., 2010; Classen et al., 2011; Dhungana et al., 2010; Juodisius et al., 2019; Kästner et al., 2009; Mendonca et al., 2009; Pohl et al., 2005; Schmid et al., 2018], mit deren Hilfe Feature-Modelle festgehalten werden können. Um eine allgemein verwendbare Sprache für Feature-Modellierung zu entwickeln, führten Sundermann et al. [2021] eine Umfrage durch. Sie untersuchten dabei deren Präferenzen in Hinblick auf Funktionalitäten von Modellierungssprachen für Feature-Modelle. Dabei ermittelten sie, dass 90 Prozent der Umfrageteilnehmer die Unterstützung von Feature-Attributen als eine gewünschte Funktionalität ansehen.

Feature-Attribute finden in erweiterten Feature-Modellen Anwendung. Durch sie können zusätzliche Informationen zu den Features festgehalten werden [Benavides et al., 2005]. So kann zu jedem Feature eines Webservers erfasst werden, welche Latenz durch das jeweilige Feature entsteht. Die Analyse von Feature-Modellen mit Attributen ist jedoch nicht trivial. Um Feature-Modelle auszuwerten, werden die Modelle in Formeln überführt und mittels Solvern erfüllende Belegungen der Formel ermittelt [Batory, 2005; Benavides et al., 2010]. Feature-Modelle ohne Feature-Attribute werden dabei in Formeln in Aussagenlogik überführt. Um Analysen auf diesen Formeln durchzuführen, können SAT-Solver verwendet werden [Apel et al., 2013].

Für Analysen von Feature-Modellen mit Feature-Attributen eignen sich SAT-Solver nur eingeschränkt: Die Feature-Modelle lassen sich zwar zuerst in Formeln in Prädikatenlogik umwandeln, welche in Aussagenlogik überführt werden können [Eén und

Sörensson, 2006; Tamura et al., 2009], jedoch sind SAT-Solver bei der Verwendung von Feature-Attributen in ihrer Funktionsweise begrenzt [Benavides et al., 2007]. Mit SMT-Solvern existieren zudem bereits Solver, die Prädikatenlogik unterstützen [Barrett et al., 2015].

Wird Prädikatenlogik für die Generierung der Formel für ein Feature-Modell verwendet, können komplexe Constraints über Feature-Attributen gebildet werden [Benavides et al., 2005] So kann für in einem Feature-Modell für ein Webserversystem jedes Feature ein Feature-Attribut *Latenz* besitzen, deren Summe nicht einen Wert von 200 ms übersteigt. Während die meisten Constraints im Feature-Modell festgehalten werden [Apel et al., 2013], können Nutzer auch während der Konfiguration die Anforderung haben, eigene Constraints zu definieren, um zusätzliche qualitative Anforderungen zu modellieren. So kann für das Webserversystem zusätzlich noch wichtig sein, dass der Ressourcenverbrauch des Systems einen Grenzwert nicht überschreitet. Derzeit können die Nutzer während des Konfigurierens nur Features ausbeziehungsweise abwählen, aber nicht solche komplexen Constraints formulieren

Zielstellung und Beiträge der Arbeit

Das Hauptziel dieser Arbeit ist es, ein Konzept zu entwickeln, mit dem Anwender während des Konfigurierens eines Produkts zusätzliche Constraints über Feature-Attributen definieren können. Wir stellen dabei Funktionen für die Auswertung von Feature-Attributen bereit, mit deren Hilfe den Anwendern die Aggregation der Feature-Attribute erleichtert wird (Summe, Produkt, arithmetisches Mittel, Maximum, Minimum).

Um dabei den Anwendungsfall dieser Arbeit von anderen Arbeiten abzugrenzen, erstellen wir eine Klassifikation, die die Anwendungsfälle anhand des Zeitpunktes im Entwicklungsprozess von Software-Produktlinien einteilt. Hierbei legen wir der Klassifizierung die Zeitpunkte zugrunde, wann Feature-Attributen Werte zugewiesen werden und wann zusätzliche Constraints durch den Nutzer definiert werden.

Um die Tauglichkeit des Konzeptes zu überprüfen, führen wir Experimente am Beispiel eines Prototypen durch und beantworten damit unsere Forschungsfragen. Die Forschungsfragen beziehen sich dabei auf folgende Schwerpunkte:

- Wird durch unser Vorgehen die Größer der Formel für das Feature-Modell beeinflusst?
- Wirkt sich unser Vorgehen auf die Zeit, die der SMT-Solver für Analysen benötigt, aus?
- Wirken sich die Aggregationsfunktionen auf die Zeit aus, die der SMT-Solver für Analysen benötigt?

Gliederung der Arbeit

Die vorliegende Arbeit ist in sechs weitere Kapitel unterteilt. In Kapitel 2 werden die theoretischen Grundlagen dieser Arbeit erläutert. Wir gehen zunächst auf Begrifflichkeiten in Bezug auf Software-Produktlinien ein und erläutern diese. Nachfolgend erläutern wir anhand des Entwicklungsprozesses, welche Zeitpunkte für diese Arbeit

und die Klassifizierung von Anwendungsfällen relevant sind. Danach gehen wir auf die Modellierung von Software-Produktlinien ein und skizzieren dabe beispielhaft eine Software-Produktlinie für einen Webserver, mit deren Hilfe wir in den nachfolgenden Kapiteln diverse Beispiele für unsere Ausführungen aufzeigen. Am Ende des Kapitels gehen wir auf gängige Analysen ein, die während der Konfiguration von Software-Produktlinien ausgeführt werden.

Nachdem die Grundlagen erläutert sind, stellen wir in Kapitel 3 zunächst unsere Klassifikation von Anwendungsfällen vor. Wir erläutern für jeden Anwendungfall, wie wir diesen von anderen Anwendungsfällen abgrenzen, und ordnen diese Arbeit in die gefundenen Anwendungsfälle ein. Nachfolgend erläutern wir, wie wir Constraints über Feature-Attribute und deren Bestandteile formal notieren. Dabei erläutern wir das Konzept der Aggregationsfunktionen. Danach gehen wir anhand eines Anwendungsbeispiels den Prozess des Konfigurierens durch, den ein Anwender durchläuft. Nach dem Beispiel erläutern wir, wie wir Feature-Modelle und Constraints über Feature-Attributen im Kontext von SMT-Solvern umsetzen.

Unser Konzept implementieren wir beispielhaft mithilfe eines Prototypen. Die dazu durchgeführten Schritte erläutern wir in Kapitel 4. Das Kapitel ist dabei in zwei Bereiche unterteilt. Der erste Bereich beschäftigt sich mit dem Einlesen des Feature-Modells. Hierbei erklären wir die von uns gewählte Struktur für das Feature-Modell und die einzelnen Schritte des Einlesens des Modells. Im zweiten Abschnitt erläutern wir unsere Realisierung von Configuring Constraints.

In Kapitel 5 beschreiben wir daraufhin, welche Forschungsfragen wir beantworten. Anhand des Prototypen führen wir diverse Untersuchungen durch und werten die Ergebnisse aus. Nachdem wir in Kapitel 6 auf verwandte Arbeiten eingehen, fassen wir unsere Erkenntnisse in Kapitel 7 zusammen und geben einen kleinen Ausblick auf mögliche Themenstellungen für zukünftige Arbeiten.

2. Grundlagen

In diesem Kapitel werden die für das Verständnis der Arbeit benötigten Hintergrundinformationen aufgeführt und kurz erläutert. Dabei wird zuerst auf Software-Produktlinien im Allgemeinen, deren Struktur und auf die damit verbundenen Feature-Modelle eingegangen. Nachfolgend wird ausgeführt, was Solver sind und inwiefern diese für die Arbeit mit und Analyse von Software-Produktlinien relevant sind.

2.1 Software-Produktlinien

Bei der Entwicklung von Software sind die Kosten der Entwicklung ein entscheidender Aspekt bei der Wahl des Vorgehens bei der Entwicklung. Ein häufig gesehener Ansatz in Unternehmen, um die Entwicklungskosten gering zu halten, ist dabei das Prinzip, verschiedene Varianten einer gemeinsamen Codebasis für unterschiedliche Kundenanforderungen zu kopieren und anzupassen. Dies resultiert jedoch in komplett unterschiedliche neue Codebasen, wodurch die eigentlich geplante Kostenminimierung nicht zum Tragen kommen kann [Fischer et al., 2014; Krüger et al., 2016; Schmid und Verlage, 2002].

Für unterschiedliche Codebasen mit ähnlichen Code schaffen Software-Produktlinien (SPLs) Abhilfe. SPLs kombinieren Aspekte von standardisierter Software mit den Möglichkeiten von individuell entwickelter Software [Apel et al., 2013]. Sie stellen wiederverwendbare Softwareteile bereit, welche durch entsprechende Konfigurationsoptionen beim Zusammenstellen einer individuellen Variante nach den Anforderungen der Nutzer an- oder abgewählt werden können. Dadurch wird eine Form der Mass Customization erreicht [Apel et al., 2013]. Die Wiederverwendung von Quellcode reduziert dabei die Entwicklungskosten, da Quellcode meist nur an einer Stelle konzeptioniert, erstellt und gewartet werden muss. Vor allem bei einer hohen Anzahl von Varianten kommt dies zum Tragen (vgl. [Knauber et al., 2001; Krüger et al., 2016; Schmid und Verlage, 2002]).
Die Wiederverwendung von Quellcode kommt auch der Codequalität zugute. Fehler, die in Varianten erkannt werden, werden in der Codebasis der SPL eliminiert,

wodurch jede Variante der SPL und in Zukunft implementierte Features davon provitieren [Clements, 1999]. Somit muss nicht jede einzelne Variante angepasst werden, wodurch wiederum Zeit und Kosten gespart werden.

SPLs reduzieren auch die Zeit bis zum Release (*time-to-market*) für die einzelnen Varianten [Knauber et al., 2001]. Dies wird vor allem deutlich, je mehr Varianten erstellt werden sollen. Bei einer geringen Anzahl an Varianten liefert eine individuelle Entwicklung schneller Ergebnisse, da die Planung der SPLs nicht berücksichtigt werden muss. Eine SPL verringert dagegen die Entwicklungszeit vor allem bei vielen Varianten, da die Wiederverwendung von einheitlichem Code den Entwicklungsaufwand stark reduziert. Das führt auch dazu, dass mit dem gleichem Aufwand von zeitlichen oder finanziellen Ressourcen mehr Features in einer SPL als bei der separaten Entwicklung für die einzelne Varianten realisiert werden können [Clements, 1999; Knauber et al., 2001]. Ein weiterer Vorteil ist, dass das eingesetzte Personal zwischen Projekten zur gleichen SPL entsprechend der benötigten Kapazitäten transferiert werden kann [Clements, 1999]. Da sich das Personal bereits mit der SPL auskennt, ist ein flüssiger, projektübergreifender Übergang ohne großen Aufwand möglich. Bei Individualprojekten ist für neues Personal hingegen meist eine Einarbeitungszeit vonnöten, um sich mit dem Projekt vertraut zu machen.

Durch SPLs können verschiedene Vorteile für die Softwareentwicklung erreicht werden, wenn viele ähnliche Varianten erstellt werden. Viele der genannten Vorteile haben dabei direkt oder indirekt positiven Einfluss auf die Entwicklungs- und Wartungskosten.

2.2 Feature-orientierte Softwareentwicklung

SPLs erlauben beim Erstellen von Varianten durch ihre unterschiedlichen Optionen, dass speziell auf den Nutzer zugeschnittene Software erstellt wird. Die Optionen beim Erstellen der Varianten bieten dabei die Möglichkeit der Individualisierung bei Beibehaltung einer hohen Wiederverwendung. Varianten einer SPL unterscheiden sich daher in der Auswahl der Optionen. Diese Auswahl bezeichnen wir als Konfiguration. Die Optionen werden als Feature bezeichnet.

Der Feature-Begriff ist in der Literatur dabei unterschiedlich definiert. Classen et al. [2008] haben dazu einige Definitionen zusammengetragen. Diese Arbeit verwendet die Definition von Apel et al. [2013], da diese die unterschiedlichen Definitionen von Classen et al. [2008] berücksichtigt und sich dabei sich auf das Vorgehen während des Entwicklungsprozesses einer SPL konzentriert. Auf Basis dieser Definition ist ein Feature wie folgt beschrieben: „Ein Feature ist eine Eigenschaft oder ein sichtbares Verhalten eines Softwaresystems. Features spezifizieren während der Erstellung von SPLs Gemeinsamkeiten und Unterschiede von Produkten. Features helfen bei der Erstellung von SPLs bezüglich Strukturierung, Wiederverwendung und Variation in allen Bereichen des Lebenszyklus der SPL.“

Features sind durch diese Definition nicht nur ein integraler Bestandteil der eigentlichen Produktlinie, sondern strukturieren auch den gesamten Entwicklungsprozess. Um die Features und deren Abhängigkeiten untereinander darzustellen, werden Feature-Modelle (siehe 2.3) verwendet.

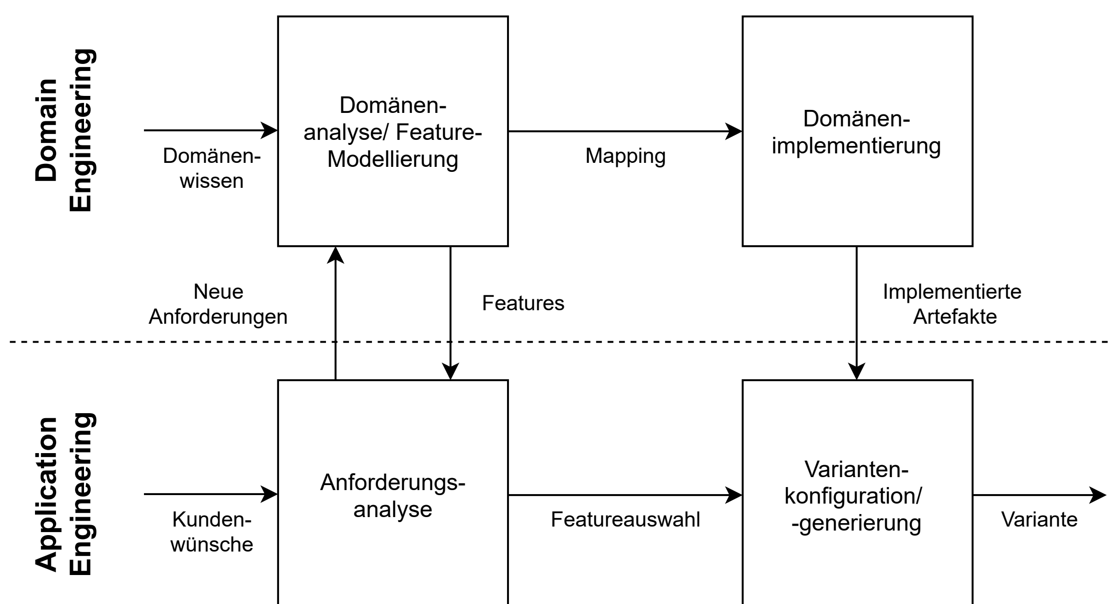

Abbildung 2.1: Entwicklungsprozess einer SPL nach Apel et al. [2013]

Um die erforderlichen Feature einer SPL zu ermitteln, muss ein für SPL angepasster Entwicklungsprozess durchlaufen werden. Dieser Prozess wird in Abbildung 2.1 dargestellt. Er besteht aus vier verschiedenen Phasen. Die Phasen lassen sich entweder der Domäne zuordnen oder beziehen sich auf eine konkrete Anwendung beziehungsweise einen konkreten Anwendungsfall. Daher lassen sich die Phasen auch dem *Domain Engineering* oder dem *Application Engineering* zuordnen.

Beim Domain Engineering liegt die Domäne dem Entwicklungsprozesses zugrunde. Eine SPL hat immer einen Gültigkeitsbereich beziehungsweise eine Domäne, die sie abdeckt. Aus dieser Domäne werden im Entwicklungsprozess Anforderungen und Wissen gesammelt, welches in Features und Feature-Modellen resultiert, die die Domäne abbilden. Die Features in den Feature-Modellen werden daraufhin implementiert und stehen als Artefakte für die Generierung beziehungsweise Konfiguration von Varianten zur Verfügung.

Durch spezielle Anforderungen an eine konkrete Variante einer SPL ist es erforderlich, dass der Entwicklungsprozess auch aus der Sicht des Application Engineering betrachtet wird. Abhängig von Kundenwünschen wird eine Anforderungsanalyse durchgeführt, in der überprüft wird, ob die Kundenwünsche durch die bestehenden Features realisiert werden können oder ob eine Neu- beziehungsweise Ummodellierung des Feature-Modells erforderlich ist. Sind die Anforderungen ausreichend erfüllt, wird die entsprechende Auswahl an Feature getroffen und anhand der Konfiguration kann nun eine Variante erstellt beziehungsweise generiert werden, welche den Kundenwünschen entspricht.

Feature werden in der Entwicklung von SPLs nicht nur für die Strukturierung von wiederverwendbaren Artefakten, sondern im gesamten Entwicklungsprozess zur Strukturierung verwendet. Anforderungen werden durch Features repräsentiert, die Features werden in Artefakten für die Variantenkonfiguration bereitgestellt, welche wiederum auf Basis einer Auswahl an Features eine Variante erzeugt. Den

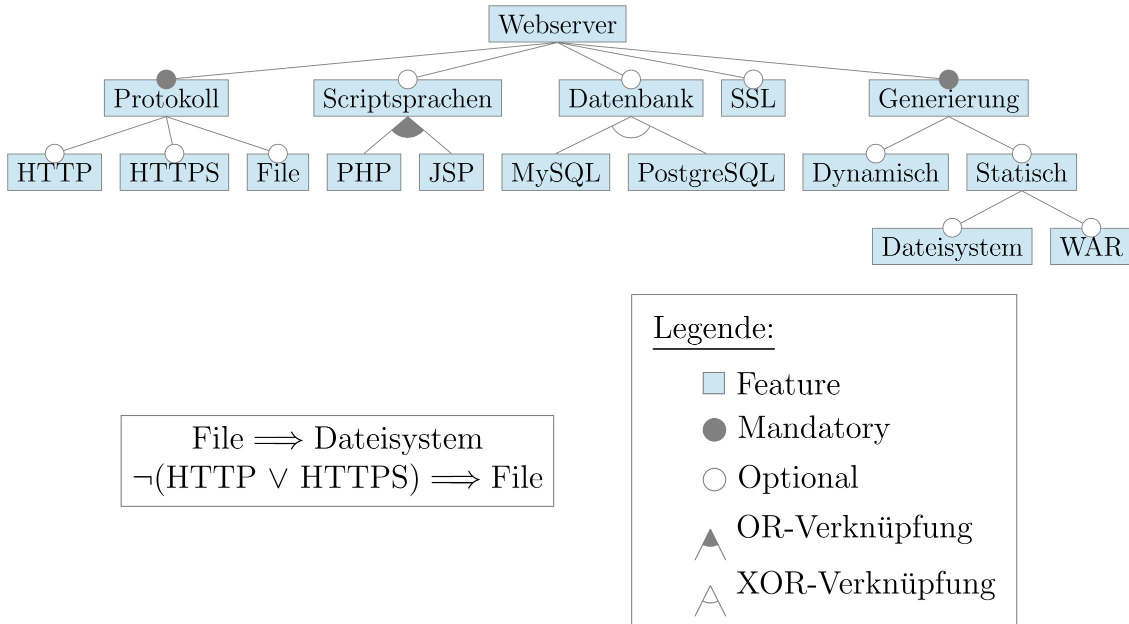

Abbildung 2.2: Feature-Diagramm eines Webservers

Entwicklungsprozess kann man dabei aus zwei Sichten betrachten. Aus der Sicht der Domäne erfolgt das Domain Engineering, welches sich auf die Domäne der SPL konzentriert. Aus der Anwendungssicht wird sich auf spezielle Anforderungen konzentriert, welche im Application Engineering in eine Variante der SPL resultieren.

2.3 Feature-Modellierung

Wurden die Features einer SPL durch den Entwicklungsprozess aus Kapitel 2.2 ermittelt, müssen sie für die spätere Verwendung festgehalten werden. Dies geschieht mittels Feature-Modellen.

2.3.1 Feature-Modelle

Formal ist ein Feature-Modell (FM) ein Tupel aus einer Menge von Features F und einer aussagenlogischen Formel $Constr$. In $Constr$ werden alle Beziehungen der Features, wie von Apel et al. [2013] beschrieben, erfasst.

$$FM = (F, Constr)$$

Anhand der aussagenlogischen Formel können automatische Analysen über dem Feature-Modell durchgeführt werden. Eine der gängigsten Analysen ist dabei die Überprüfung, ob eine Konfiguration für eine SPL valide ist. Werden Feature in der Konfiguration ausgewählt, wird ihnen der Wahrheitswert *wahr* zugewiesen. Sind sie nicht in der Konfiguration enthalten, wird ihnen der Wahrheitswert *falsch* zugewiesen. Setzt man diese Werte in die aussagenlogische Formel ein, ist diese entweder erfüllt oder führt zu einem Widerspruch. Somit kann für eine Konfiguration die Aussage getroffen werden, ob diese Konfiguration valide für die SPL ist. Auch

für Teilkonfigurationen können automatisch Auswertungen über die aussagenlogische Formel durchgeführt werden (vgl. Abschnitt 2.4).

Um die Kommunikation zwischen Entwicklern über die Feature-Modelle zu erleichtern, werden diese nicht nur formal, sondern auch grafisch in Feature-Diagrammen wie in Abbildung 2.2 dargestellt. Das Diagramm zeigt beispielhaft eine SPL für einen Webserver. Dabei werden die Beziehungen der Features untereinander durch die Modellierung in einer Baumstruktur dargestellt. Die Knoten repräsentieren die eigentlichen Feature. Features wie *Protokoll* und *Generierung* sind als *mandatory* markiert und müssen bei jeder Konfiguration enthalten sein. Andere Features wie *Datenbank* oder *SSL* sind *optional* und können nach Bedarf der Konfiguration hinzugefügt werden. Durch die Baumstruktur im Diagramm stehen Features nicht nur alleine, sondern besitzen unterordnete Features. Während bei *Protokoll* alle untergeordneten Features als optional gekennzeichnet sind, sind die Features *PHP* und *JSP* durch eine *Oder*-Verknüpfung mit dem Feature *Scriptsprachen* verbunden. Ist damit das Feature *Scriptsprachen* ausgewählt, muss mindestens ein untergeordnetes Feature ausgewählt werden. Die untergeordneten Features von *Datenbank* sind dagegen mit einer XOR-Verknüpfung mit dem Feature *Datenbank* verknüpft. Hier muss genau ein Feature als *Alternative* ausgewählt werden, sobald *Datenbank* in der Konfiguration enthalten ist. Nun kann es zu dem Fall kommen, dass ein Feature in einem Teilbaum des Diagramms mit einem Feature aus einem anderen Teilbaum in Beziehung steht. Beispielsweise erfordert das *Datei*-Protokoll Zugriff auf das Feature *Dateisystem*. Dies kann durch *Cross-Tree-Constraints* realisiert werden, welche die Abhängigkeiten durch aussagenlogische Formeln in Textform darstellen. Im Beispiel tritt dadurch auch der Fall ein, dass mindestens ein Protokoll erforderlich ist, obwohl alle untergeordneten Features von *Protokoll* als optional markiert sind.

2.3.2 Feature-Attribute

Abhängig von der Domäne kann es notwendig sein, dass die Feature-Modelle weitere Informationen beinhalten als nur den Namen des Features. Um diese zusätzliche Informationen über die Features zu erfassen, haben sich Feature-Attribute etabliert. Feature-Attribute sind keine eigenen Feature, da sie selbst keine Funktionalität bereitstellen, sondern stellen eine messbare Charakteristik des Features dar [Benavides et al., 2005]. Auf Basis dieser Feature-Attribute ist es möglich, eine Konfiguration nicht nur nach der Auswahl der Features zu betrachten, sondern auch weiterführende Analysen nach der Sinnhaftigkeit der Konfiguration durchzuführen. So ist beispielsweise auf Mikrocontrollern die Hardware-Leistung stark beschränkt, wodurch es erforderlich ist, dass die Features die Kapazität des Controllers nicht überschreiten. Für eine Konfiguration kann somit überprüft werden, inwiefern sie Leistung des Mikrocontrollers in Anspruch nimmt.

Um Feature-Attribute zu berücksichtigen, muss jedoch die formale Definition des Feature-Modells angepasst werden. Um Feature-Attribute in den Constraints verwenden zu können, muss das Tupel FM um eine weitere Menge $Attr$ erweitert, welche die Feature-Attribute enthält. Ein Attribut ist dabei eine Funktion, welche einem Feature einen Wert aus der Domäne des Attributs zuweist [Benavides et al., 2010]. Somit lautet das Tupel

$$FM = (F, Constr, Attr).$$

In der graphischen Repräsentation des Feature-Modells werden aufgrund der Lesbarkeit keine separaten Elemente hinzugefügt. Stattdessen wird eine Tabelle der Feature-Attribute zusätzlich zum Diagramm erstellt und für die Kommunikation bereitgestellt.

Feature	Latenz	MAnf	Ausfall
Webserver	10	1000000	-
Scriptsprachen	20	25000	-
Datenbank	20	-	-
MySQL	-	300000	1
PostgreSQL	-	50000	1.1
Dateisystem	20	20000	-
WAR	20	400000	2
SSL	10	500000	-

Tabelle 2.1: Beispielhafte Werte für die Attribute *Latenz*, *MAnf*, *Ausfall*

Für unser Beispiel aus Abbildung 2.2 führen wir das Feature-Attribut *Latenz* ein. Das Attribut beschreibt für die Features die durchschnittliche Verzögerung der Anfragen, die durch das Auswählen des Features zusätzlich zu erwarten ist. Eine Auflistung beispielhafter Werte für die SPL aus Abbildung 2.2 wird in Tabelle 2.1 aufgeführt. Enthält ein Feature ein Feature-Attribut nicht, wird dies durch das Zeichen „-“ symbolisiert. Aufgrund des Feature-Modells ist im Beispiel abhängig von der Konfiguration von einer durchschnittlichen Latenz zwischen 30 und und 100 Millisekunden auszugehen. Weiterhin sind in Tabelle 2.1 die Werte für das Feature-Attribut *MAnf* dargestellt. Dabei handelt es sich um die maximale Anzahl der Anfragen, die das jeweilige Feature pro Minute unterstützt. Auch wird die zu erwartende prozentualen Ausfallwahrscheinlichkeit der Features *Ausfall* in einem Jahr in der Tabelle dargestellt.

Konstante Feature-Attribute

Feature-Attribute können während des SPL-Entwicklungsprozesses sowohl im Domain- als auch im Application-Engineering definiert werden. So werden die meisten Attribute im Domain-Engineering erfasst [Montagud et al., 2012]. Zhang et al. [2011] verwenden beispielsweise die im Domain-Engineering definierten Attribute, um beim Konfigurieren eine Entscheidungshilfe zur Anforderungserfüllung bereitzustellen. Auch Ochoa et al. [2015] legen ihrer Analyse die im Feature-Modell enthaltenen Attribute zugrunde, um unternehmensrelevante Daten auszuwerten. Die Werte der Attribute, die dabei festgelegt werden, gelten für alle Varianten der SPL und unterscheiden sich von Variante zu Variante nicht. Da diese Attribute für alle Varianten einer SPL somit identisch sind, verwenden wir für diese Attribute den Begriff *konstante Attribute*. Diese Attributewerte sind durch das Feature-Modell vorgegeben und in Hinblick auf eine SPL konstant. Im Beispiel aus Tabelle 2.1 ist das Attribut *Latenz* ein konstantes Feature-Attribut, da es im Domain-Engineering festgelegt wird und für alle Varianten identisch ist.

Dynamische Feature-Attribute

Attribute, denen im Application-Engineering Werte zugewiesen werden, können für jede Alternative der SPL unterschiedlich sein. So ermitteln Zanardini et al. [2016] auf Basis von Varianten der SPL den Ressourcenverbrauch. Auch Attribute wie Kardinalitäten werden während des Konfigurierens, und damit im Application-Engineering, angewendet. Da diese Attribute in Hinblick auf eine SPL veränderbar sind, verwenden wir den Begriff *dynamisch*. Sie werden in der Regel erst bei der Auswahl der Features oder beim Konfigurieren der Variante erfasst. Für unser Beispiel kann ein Kunde den Wunsch haben, dass mehrere Datenbanken unterstützt werden. Beim Konfigurieren gibt der Kunde nun an, dass er beispielsweise zwei Datenbanken unterstützen will. In der finalen Variante werden dadurch zwei Einstellungen für die zwei Datenbanken hinterlegt, wo der Kunde nun die Verbindungsdaten hinterlegen kann.

2.3.3 Constraints

Ähnlich wie Feature-Attribute, die sowohl im Domain- als auch im Application Engineering initialisisert werden können, können auch Constraints zu unterschiedlichen Zeitpunkten im SPL-Entwicklungsprozesses definiert werden. Da die Constraints im Domain-Engineering bei der Modellierung definiert werden, wollen wir diese Model-Constraints nennen. Die Constraints, die im Application-Engineering erfasst werden, werden beim Konfigurieren der Varianten der SPL erfasst, weswegen wir diese Configuring Constraints nennen. Sowohl Model-Constraints als auch Configuring Constraints haben dabei die gleiche Funktion: die Filterung der möglichen Konfigurationen.

Zu den Model-Constraints zählen dabei alle Constraints, die die Beziehungen der Feature untereinander abbilden, sowie alle Cross-Tree-Constraints. Diese Constraints werden für alle Varianten der SPL definiert und definieren damit den Anwendungsfall, in dem alle Varianten angesiedelt sind.

Die Configuring Constraints zählen im Vergleich zu den Model-Constraints vor allem für die Realisierung eines spezifischen Anwendungsfalls für eine Variante der SPL. Da sie beim Konfigurieren erfasst werden, schränken sie die Feature-Auswahl für eine Konfiguration ein. Sie stellen zusätzliche Einschränkungen des Nutzers dar, die dieser während des Konfigurierens zu beachten hat. Benavides et al. verwenden für diese Art der Constraints den Begriff Filter.

2.4 Feature-Modell-Analyse

Die Auswertung von Feature-Modellen kann abhängig von der SPL sehr komplex werden. So besitzt beispielsweise der Linux-Kernel über 10000 Features [Tartler et al., 2011]. Um die Anwender bei der Auswertung und Konfiguration von SPLs zu unterstützen, haben sich einige Analysen etabliert [Apel et al., 2013].

1. Valide Konfiguration: Die gängigste Analyse ist dabei die Überprüfung, ob eine ausgewählte Konfiguration valide ist. Dabei wird überprüft, ob eine Auswahl an Features durch die Beschränkungen der SPL zulässig ist.

$$Webserver\ (1)$$
$$Protokoll \Leftrightarrow Webserver\ (2)$$
$$HTTP \Rightarrow Protokoll\ (3)$$
$$HTTPS \Rightarrow Protokoll\ (3)$$
$$File \Rightarrow Protokoll\ (3)$$
$$Scriptsprachen \Rightarrow Webserver\ (3)$$
$$(PHP \vee JSP) \Leftrightarrow Scriptsprachen\ (5)$$
$$Datenbank \Rightarrow Webserver\ (2)$$
$$((MySQL \vee PostgreSQL) \Leftrightarrow p) \wedge ((\neg MySQL \wedge PostgreSQL) \wedge (MySQL \wedge \neg PostgreSQL))\ (4)$$
$$Generierung \Leftrightarrow Webserver\ (2)$$
$$Dynamisch \Rightarrow Generierung\ (3)$$
$$Statisch \Rightarrow Generierung\ (3)$$
$$Dateisystem \Rightarrow Statisch\ (3)$$
$$WAR \Rightarrow Statisch\ (3)$$
$$File \Rightarrow Dateisystem\ (6)$$
$$\neg(HTTP \vee HTTPS) \Rightarrow File\ (6)$$

Abbildung 2.3: Formalisierung des Beispiels aus Abbildung 2.2

2. Dead-Features: Bei der Analyse nach toten Features wird überprüft, ob es Features gibt, die niemals ausgewählt werden können. Dies dient zur Erkennung von unerreichbarem Quellcode und verbessert somit die Wartbarkeit der SPL.
3. Core-Features: Wenn Core-Features in einer SPL gesucht werden, wird nach allen Features gesucht, die in jeder Variante enthalten sein muss. Dies dient unter anderem zur Erkennung der kleinsten erforderlichen Konfiguration und kann Hinweise auf die Variabilität der SPL liefern. Je Features Core-Features sind, desto geringer ist die Variabilität der SPL.
4. Decision Propagation: Bei Decision Propagation [Krieter et al., 2018] oder Constraint Propagation [Apel et al., 2013] wird beim Konfigurieren einer Alternative automatisiert ausgewertet, welche Features aufgrund der derzeitigen Selektion zwingend enthalten sein müssen oder nicht mehr enthalten sein können. Dabei wird der Nutzer durch die Limitierung der Auswahlmöglichkeiten daran gehindert, eine invalide Konfiguration zu erstellen.

Um Analysen auf dem Feature-Modell auszuführen, werden die Feature-Modelle und Analysen in Formeln formalisiert.

(1) Bei einem Feature-Modell wird dabei am Wurzelknoten begonnen. Da dieser in jeder Variante der SPL enthalten sein muss, wird dieses Feature der noch leeren Formel als positives Literal hinzugefügt. Ausgehend vom Wurzelknoten werden nun alle anderen Features durchlaufen und über Konjunktionen der Formel hinzugefügt.

(2) Besteht eine Mandatory-Verknüpfung zwischen dem übergeordneten Feature p und untergeordneten Feature f, wird eine Formel $f \Leftrightarrow p$ hinzugefügt.

(3) Sind das übergeordnete Feature p und das untergeordnete Feature f mit einer optionalen Verknüpfung verknüpft, wird der Formel für das Feature-Modell die Formel $f \Rightarrow p$ hinzugefügt.

(4) Gibt es zu dem übergeordneten Feature p mehrere untergeordnete Features f_1 bis f_n, die Alternativen zueinander sind, wird der Formel für das Feature-Modell die Formel $((f_1 \vee \cdots \vee f_n) \Leftrightarrow p) \wedge \bigwedge_{i<j} \neg(f_i \wedge f_j)$ hinzugefügt.

(5) Sind die Features f_1 bis f_n dagegen dem Feature p untergeordnet und mit diesem mittels einer Oder-Verknüpfung verknüpft, wird die Formel für das Feature-Modell um die Formel $(f_1 \vee \cdots \vee f_n) \Leftrightarrow p$ erweitert.

(6) Alle Cross-Tree-Constraints müssen nun noch hinzugefügt werden. Da diese meist als aussagenlogische Formel vorliegen, können sie der Formel für das Feature-Modell mittels Konjunktionen hinzugefügt werden.

Auf Basis dieses Vorgehens lässt sich auch das Feature-Modell aus Abbildung 2.2 wie in Abbildung 2.3 formalisieren. Dabei muss beachtet werden, dass aufgrund der Lesbarkeit die Konjunktionen der einzelnen Formeln weggelassen wurden und diese noch in der finalen Formel *Constr* hinzugefügt werden müssen.

Die Analysen, die auf den formalisierten Feature-Modellen ausgeführt werden, lassen sich auch in aussagenlogische Formeln umwandeln.

1. Valide Konfiguration: Damit eine Konfiguration als valide gilt, muss die aussagenlogische Formel *Constr* erfüllt sein, wenn jedem ausgewählten Feature der Wert *wahr* und jedem nicht-ausgewählten Feature der Wert *falsch* zugewiesen wird. Ist im Beispiel aus Abbildung 2.3 das Feature *Webserver* nicht ausgewählt, würde dies zu einem Widerspruch führen und die Konfiguration ist nicht valide (vgl. Apel et al. [2013]).
2. Dead-Features: Für jedes Feature f wird überprüft, ob es eine valide Konfiguration gibt, die die Formel $Constr \wedge f$ erfüllt. Wird keine solche gefunden, ist das Feature f in keiner validen Konfiguration für die SPL enthalten. Wenn dem Beispiel aus Abbildung 2.3 die Bedingung $MySQL \Rightarrow PostgreSQL$ hinzugefügt wird, würde dies das Feature *MySQL* als totes Feature markieren.
3. Core-Features: Für jedes Feature f wird überprüft, ob es eine valide Konfiguration gibt, die die Formel $Constr \wedge \neg f$ erfüllt. Wird keine solche gefunden, ist das Feature f in jeder validen Konfiguration für die SPL enthalten. Für das Beispiel gelten *Webserver*, *Protokoll* und *Generierung* als Core-Features.
4. Decision Propagation: Während des Konfigurierens wird für eine Selektion von Features und das Feature f überprüft, ob die aussagenlogischen Formel $Constr \wedge f$ nicht erfüllbar ist. Ist dies der Fall, kann dass Feature in einer validen Konfiguration nicht mehr ausgewählt werden. Ist stattdessen die Formel $Constr \wedge \neg f$ nicht erfüllbar, muss das Feature f in einer validen Konfiguration enthalten sein. Werden im Beispiel die Core-Features ausgewählt, wäre durch Decision Propagation auch das Feature *File* und somit auch das Feature *Dateisystem* in der Konfiguration enthalten.

Alle genannten Analysen greifen auf das Prinzip der Erfüllbarkeit von aussagenlogischen Formeln zurück. Dabei ist die Erfüllbarkeit ein NP-vollständiges Problem, wodurch es keinen Algorithmus gibt, der diese Entscheidung in nichtdeterministisch in Polinomialzeit teffen kann. Trotz dessen hat die Praxis gezeigt, dass auch bei komplexen Formeln schnell und effizient gute Ergebnisse erzieht werden, wenn entsprechende Solver verwendet werden [Wasowski et al., 2009]. Gängige Solver sind dabei unter anderem SAT-Solver und SMT-Solver [Apel und Kästner, 2009]. SAT-Solver liefern auf Basis einer aussagenlogischen Formel Konfigurationen zurück, die diese Formel erfüllbar machen. Dabei weisen sie den booleschen Variablen solange Wahrheitswerte zu, bis es eine erfüllende Belegung gibt oder bis alle Variablen behandelt wurden und keine erfüllende Belegung gefunden werden konnte [Wasowski et al., 2009]. Eine Implementation eines SAT-Solvers in Java ist der Solver SAT4J [Berre und Parrain, 2010], der für viele Formeln innerhalb von Millisekunden Ergebnisse zurückliefert [Apel et al., 2013].

Einige Probleme erfordern mehr Funktionalität als Wahrheitswerte alleine bereitstellen können. Folglich können hier SAT-Solver nur bedingt Abhilfe schaffen. Daher haben sich SMT-Solver etabliert. SMT (Satisfiability modulo theories) sind eine Erweiterung von SAT und erlauben nicht nur die Verwendung von aussagenlogischen Formeln, sondern erweitern diese Funktionalität um diverse arithmetische Operationen [Moura und Bjørner, 2008]. So erweitern sie den Funktionsbereich unter anderem um Ungleichungen und Gleichungen, Summen und Produkte [Barrett et al., 2015] Durch die Verwendung von SMT-Solvern zur Analyse von Feature-Modellen es möglich, enthaltene Feature-Attribute nach Qualitätskriterien in die Analyse mit einzubeziehen. So wäre es am Beispiel des Webservers möglich, jede Variante der SPL zu ermitteln, welche eine Latenz von weniger als 50 Millisekunden aufweist.

Da sich die Auswertungen von Constraints über Attributen gut auf die Funktionen von SMT-Solver abbilden lassen, verwenden wir in dieser Arbeit SMT-Solver für die Auswertung der Feature-Modelle und die dazugehörigen Analysen. Im speziellen konzentrieren wir uns auf die Funktionen der Gleichheit, Ungleichheit, der Summen- und Produktbildung.

3. Konzept

Nachdem wir im vorigen Kapitel Feature-Modelle, SPLs und Solver vorgestellt haben, wenden wir Feature-Attribute nun bei den automatischen Analysen an. Dazu gehen wir zunächst auf mögliche Anwendungsfälle von Constraints über Feature-Attributen ein. Nachfolgend definieren wir den den Begriff der Configuration-Constraints genauer. Im Anschluss zeigen wir eine mögliche Abbildung der Configuration-Constraints für die Verwendung mit SMT-Solvern auf. Wir konzentrieren uns dabei auf numerische Datentypen wie ganze oder reelle Zahlen, da wir diese durch arithmetische Operationen auswerten können.

3.1 Motivation

Im Entwicklungsprozess von SPLs werden Constraints zu verschiedenen Phasen im Prozess definiert (vgl. Unterabschnitt 2.3.3). Zum einen werden sie während des Modellierens des Feature-Modells erfasst. Sie werden damit gleichermaßen für alle Varianten der SPL gleichzeitig festgelegt. In Abbildung 2.2 werden Cross-Tree-Constraints unterhalb des Modells als Formeln dargestellt. Zusätzlich dazu werden die Beziehungen der Features untereinander in Formeln dargestellt. Zum anderen können Constraints separat beim Konfigurationsprozess definiert werden. Dadurch werden spezifische Anforderungen an die konkret zu konfigurierende Variante der SPL gewährleistet, ohne dass das Feature-Modell angepasst werden muss, da diese Constraints nur für diese spezielle Variante einzuhalten sind. Cross-Tree-Constraints können also nach dem Zeitpunkt ihres Erfassens klassifiziert werden.

Feature-Attribute werden beim Modellieren deklariert und im Feature-Modell festgehalten. Dabei werden den Features die Attribute zugeordnet und mit einem Namen versehen. Beim Webserver hat das Feature *Datenbank* das Feature-Attribute *Latenz* zugewiesen bekommen. Für das Feature wurde daher ein Feature-Attribut deklariert. Aber obwohl die Feature-Attribute während des Modellierens deklariert werden, können ihnen ihre Werte zu verschiedenen Phasen im Entwicklungsprozess zugewiesen werden. Wie in Unterabschnitt 2.3.2 beschrieben, kann dies während der Erstellung des Feature-Modells oder während des Konfigurierens erfolgen. Feature-Attribute

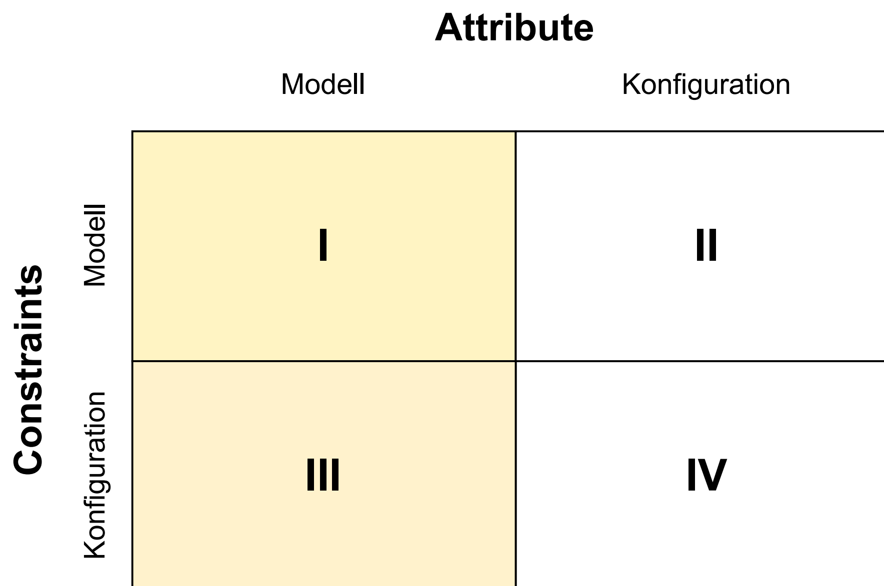

Abbildung 3.1: Anwendungsfälle kategorisiert nach Constraints und Feature-Attributen (Gelb: Anwendungsfälle der Arbeit, Weiß: Weitere Anwendungsfälle)

können dementsprechend nach dem Zeitpunkt klassifiziert werden, wann ihnen ein Wert zugewiesen wird. In Abbildung 3.1 wird dies zusammen mit der Klassifizierung der Constraints dargestellt, um die verschiedenen Anwendungsfälle von Constraints über Feature-Attributen zu unterscheiden. Dabei stellt jeder Quadrant einen eigenen Anwendungsfall dar, die wir in den nachfolgenden Unterabschnitten einzeln erläutern, um sie voneinander zu unterscheiden.

3.1.1 Model-Attribute und Model-Constraints

Wenn bei der Entwicklung des Feature-Modells sowohl die Werte der Feature-Attribute als auch alle Constraints festgelegt werden, befinden wir uns im ersten Quadranten und damit beim ersten der vier Anwendungsfälle. Dieser Fall ermöglicht die Überprüfung des Feature-Modells: Ist es für die SPL aus Abbildung 2.2 beispielsweise wichtig, dass die aufsummierte Latenzen eine maximale Latenz von 20 Millisekunden nicht überschreiten, kann bereits beim Modellieren mittels Void-Analyse festgestellt werden, ob das Modell eine valide Konfiguration erzeugt. Somit ist es möglich, präzise die Anwendungsfälle für die SPL zu definieren und den *Scope* der SPL festzulegen [Apel et al., 2013].
Speziell bei Hardware-beschränkten Zielsystemen kann dies wichtig werden: So kann beispielsweise mittels der Analysen von Zanardini et al. [2016] ermittelt werden, wie groß der Ressourcenverbrauch pro Feature ist und das Feature kann mit Feature-Attributen inklusive Werten versehen werden. Bei Geräten, wo der Speicher stark limitiert ist, wird somit gewährleistet, dass die SPL diesen nur zu einem festgelegten Prozentsatz oder bis zu einer festgelegten Grenze ausreizen kann. So kann festgelegt werden, dass die gesamte Speicherverbrauch, also die Summe des Features *Speicherverbrauch* für alle ausgewählten Features den Grenzwert von beispielsweise 500MB nicht überschreitet.

3.1.2 Configuration-Attribute und Model-Constraints

Um den Ressourcenbedarf der Features zu bestimmen, analysieren Zanardini et al. [2016] eine minimale Konfiguration mit dem zu analysierenden Feature. Sie generieren dabei eine Variante der SPL und berechnen deren Ressourcenbedarf. Da dies im Application Engineering angesiedelt ist, argumentieren wir, dass die durch die Analysen ermittelten Werte für die Feature-Attribute nicht beim Modellieren, sondern während des Konfigurierens festgelegt werden. Damit befinden wir uns nicht mehr im ersten Quadranten, da hier die Werte für die Feature-Attribute beim Modellieren festgelegt werden.
Tatsächlich können die Anwendungsfälle aus Abbildung 3.1 ineinander verschwimmen und miteinander kombiniert werden. Zanardini et al. [2016] erlauben mit ihrer Arbeit nicht nur die Unterstützung für den ersten, sondern auch für den zweiten Anwendungsfall. Bei diesem werden die Constraints im Modell während des Modellierens festgelegt, während die Feature-Attribute nur deklariert werden. Die Wertzuweisung zu den Feature-Attributen erfolgt während des Konfigurierens. Zusätzlich zum Scoping besteht in diesem Anwendungsfall die Möglichkeit, dass Nutzer zusätzliche Parameter im Konfigurationsprozess angeben können, um ihre speziellen Anforderungen zu erfüllen.
Ein weiteres Beispiel für den zweiten Anwendungsfall sind Kardinalitäten. Während des Konfigurierens können mehrere Instanzen von Features in die Konfiguration eingebunden werden. Nehmen wir an, dass im Beispiel von Abbildung 2.2 mehrere Datenbanken unterstützt werden sollen. Mittels Kardinalitäten wird dies gewährleistet werden, indem dem Feature eine Kardinalität von mindestens 1 ohne Obergrenze zugewiesen wird. Haben die Entwickler der SPL jedoch festgestellt, dass mehr als zwei Datenbanken nicht notwendig sind oder nicht unterstützt werden können, können sie ein entsprechendes Constraint im Feature-Modell hinzufügen.

3.1.3 Model-Attribute und Configuration-Constraints

Wenn die Attribute während des Modellierens deklariert werden und ihnen zudem Werte zugewiesen werden, aber keine zusätzlichen Constraints über diesen Attributen definiert werden, befinden wir uns im dritten Quadranten. Hier werden die Constraints erst während des Konfigurierens durch den Nutzer festgelegt, der somit an seine Anforderungen spezialisierte Produkte erhält. Benavides et al. [2005] bezeichnen die durch den Nutzer festgelegten Constraints beispielsweise als Filter, da sie die Anzahl der möglichen Konfiguration einschränken.
Wir nehmen für unser Beispiel aus Abbildung 2.2 an, dass zwei Kunden die SPL bei sich einsetzen wollen, jedoch unterschiedliche Anforderungen haben. Beim ersten Kunden sei es wichtig, dass die Antwortzeiten des Webservers gering sind. Die Latenz muss also möglichst klein sein. Hierzu definiert dieser Kunde beim Konfigurieren, dass die die aufsummierten Latenzen aller ausgewählten Features einen Wert von 100ms nicht überschreiten soll. Für den zweiten Kunden sei die Antwortzeit nicht relevant. Stattdessen sei es für ihn wichtig, dass die von ihm erstellten Produkte mindestens 30000 Anfragen pro Minute unterstützen. Daher definiert der zweite Nutzer bei der Konfiguration dies als Constraint. Im Beispiel aus Abbildung 2.2 schränkt die Anforderung des zweiten Nutzers die Funktionalität des Produktes auf Ausprägungen ein, die die Features *Scriptsprachen* und *Dateisystem* nicht enthalten.

Diese Features unterstützen nur 20000 beziehungsweise 25000 Anfragen pro Minute. Anhand des Beispiel lässt sich erkennen, dass der große Unterschied dieses Quadranten zu den bisher aufgeführten darin liegt, dass die Nutzer während der Konfiguration eine hohe Kontrolle über die Ausprägung der SPL besitzen, ohne dass das Feature-Modell angepasst werden muss. Es gibt eine strikte Trennung zwischen generellen Anforderungen (Constraints festgelegt im Feature-Modell) und spezifischen Anforderungen (Constraints festgelegt durch die Nutzer während der Konfiguration). Dies führt hauptsächlich dazu, dass das Feature-Modell sauber gewartet werden kann, da spezifische Anforderungen wohlüberlegt in das Modell übernommen werden können, jedoch nicht müssen.

3.1.4 Configuration-Attribute und Configuration-Constraints

Der letzte und vierte Quadrant aus Abbildung 3.1 beschreibt den Anwendungsfall, dass die Werte der Feature-Attribute und die darauf aufbauenden Constraints während des Konfigurierens festgelegt werden. In diesem Fall sind sowohl die Werte der Feature-Attribute als auch die Constraints über diesen nicht zum Zeitpunkt des Modellierens bekannt. Da SPLs jedoch in Bezug auf eine Domäne entwickelt beziehungsweise modelliert werden, tritt dieser Fall in Reinform eher selten auf, da durch die Abgrenzung der SPL bereits Constraints gebildet und Feature-Attribute mit Werten versehen werden.
Der Anwendungsfall liegt jedoch vor, wenn die Ausprägungen einer SPL komplett durch externe Vorgaben wie durch gesetzliche oder firmeninterne Regelungen bestimmt werden. Beim Modellieren wird die SPL dabei so definiert, dass die Nutzer beim Konfigurieren gänzlich frei bei der Gestaltung ihrer Konfiguration sind. Die Nutzer haben somit die Aufgabe, die ihnen vorliegenden Regelungen und Bestimmungen in Werten für die Feature-Attribute und Constraints abzubilden.

3.1.5 Anwendungsfall der Arbeit

Wie bereits erwähnt, lassen sich die Anwendungsfälle nicht immer klar voneinander abgrenzen. Dies ist auch der Fall in dieser Arbeit. In dieser Arbeit konzentrieren wir uns auf den markierten dritten Quadranten, da sich keine anderen Arbeiten mit diesem befassen. Das Ergebnis der Arbeit kann aber auch für den ersten Quadranten, also der Definition der Constraints über Feature-Attributen und der Zuweisung der Werte für diese Attribute während des Modellierens, Anwendung finden. Wir treffen zur Abgrenzung des Anwendungsfalls dieser Arbeit folgende Festlegungen.

- Alle Feature-Attribute, die in den Constraints verwendet werden, sind im Feature-Modell definiert und es wurde ihnen während des Modellierens ein Wert zugewiesen. Damit handelt es sich bei den Feature-Attributen um Konstanten.
- Wir erlauben es Nutzern, während des Konfigurierens Constraints zu definieren, die nicht bereits im Feature-Modell definiert sind, um spezifische Anforderungen an die SPL abzubilden. Diese können dabei auf den Konstanten für die Feature-Attribute zurückgreifen.
- Eine Konfiguration muss sowohl die Constraints des Feature-Modells als auch die zusätzlichen Constraints, die durch den Nutzer festgelegt wurden, erfüllen, damit sie als valide gilt.

- Während des Konfigurierens werden keine Wertzuweisungen zu Feature-Attribute durch den Nutzer vorgenommen.

Durch diese Einschränkungen konzentrieren wir uns in dieser Arbeit auf den dritten Quadranten (vgl. Abbildung 3.1), da die Feature-Attribute während des Modellierens sowohl deklariert als auch initialisiert werden, Nutzer jedoch zusätzliche Constraints während des Konfigurierens festlegen können.

3.2 Formale Definition

In Abschnitt 3.1 beschreiben wir anhand von Beispielen die unterschiedlichen Anwendungsfälle für Constraints über Feature-Attribute. Diese haben wir bisher allgemein formuliert. Wir benötigen nun eine Möglichkeit, um diese auch maschinell zu verarbeiten und zu analysieren. In diesem Abschnitt stellen wir daher die Syntax vor, der die Constraints folgen.

3.2.1 Syntax

Um die Constraints über Feature-Attribute abzubilden und formal zu beschreiben, verwenden wir Prädikatenlogik, da wir sowohl Funktionen benötigen als auch atomare Formeln um Vergleiche erweitern. Quantifizierung über Quantoren werden in den Formeln nicht verwendet. Jedes Constraint wird durch eine eigene Formel repräsentiert.

Formel. Eine Formel besteht entweder aus einer atomaren Formel, der Verbindung zweier Formeln durch Konjunktion ($\wedge$), Disjunktion ($\vee$), Implikation ($\Rightarrow$) oder Bi-Implikation ($\Leftrightarrow$), oder ist selbst die Negation ($\neg$) einer anderen Formel. Beispielsweise sind damit folgende Formeln möglich, wenn f_1 bis f_n selbst Formeln sind:

$$f_1 \vee f_2 \Rightarrow f_3$$
$$f_1 \Leftrightarrow f_2$$
$$\neg f_3 \wedge f_4$$

Atomare Formel. Atomare Formeln sind die Wahrheitswerte wahr ($\top$) und falsch ($\bot$) sowie ein Prädikat zweier Terme mittels Gleichheit ($=$), Ungleichheit ($<$ oder $>$) oder der Kombination dieser beiden ($\leq$ oder $\geq$). Auch Features werden in den Constraints als atomare Formeln behandelt, wobei die Variablen für die Features die Werte wahr oder falsch annehmen können. Diese haben den gleichen Namen wie die dazugehörigen Features. Für das Beispiel aus Abbildung 2.2 sind also die Features *Datenbank* und *Statisch* möglich, wodurch sich wie bei normalen aussagenlogischen Modellen folgende Formeln bilden lassen:

$$Datenbank \wedge Statisch$$
$$Statisch \Rightarrow Datenbank$$
$$\neg Datenbank \vee Statisch$$

Feature	Konstante	Wert der Konstante
Webserver	$Latenz(Webserver)$	10
Dateisystem	$Latenz(Dateisystem)$	20
WAR	$Latenz(WAR)$	20
SSL	$Latenz(SSL)$	10
Datenbank	$Latenz(Datenbank)$	20
Scriptsprachen	$Latenz(Scriptsprachen)$	20

Tabelle 3.1: Konstanten für die Werte des Feature-Attributs *Latenz* basierend auf den Werten aus Tabelle 2.1

Term. In den Formeln der Constraints können Terme verwendet werden. Die Terme können dabei Variablen, Konstanten oder Funktionen angewendet auf eine beliebige Anzahl an Termen sein. Da in unserem Anwendungsfall die Werte für Feature-Attribute während des Modellierens zugewiesen werden, betrachten wir sie als Konstanten. Sie bestehen aus dem Namen des Features, zu dem sie gehören, und dem Namen des Attributs.

Beispiel. Für die SPL aus Abbildung 2.2 können unter anderem die Konstanten *Latenz(Datenbank)* oder *Latenz(Webserver)* verwendet werden. Dabei bekommt *Latenz(Datenbank)* entsprechend Tabelle 2.1 den Wert 20 ($Latenz(Datenbank) = 20$) und *Latenz(Webserver)* den Wert 10 ($Latenz(Webserver = 10$) zugewiesen. Weitere mögliche Konstanten für das Beispiel aus Abbildung 2.2 sind in Tabelle 3.1 dargestellt.

Für den Anwendungsfall IV aus Abbildung 3.1, also dem Festlegen der Werte für die Feature-Attribute während der Konfiguration, sind die Werte der Feature-Attribute als Variablen zu behandeln.

Funktion. Funktionen ordnen gegebenen Parameter einen Wert zu. Dabei liefern sie einen Wert zurück, der in den Constraints ausgewertet werden kann. Gibt es also die Funktionen P_1 bis P_n, die jeweils ein Feature als Parameter übergeben bekommen und eine natürliche Zahl zurückliefern, können für unser Beispiel aus Abbildung 2.2 unter anderem folgende Constraints gebildet werden:

$$P_1(Datenbank) < 12$$
$$P_1(Datenbank) \leq P_2(Dateisystem)$$
$$P_3(Scriptsprachen) = P_3(SSL)$$
$$P_4(Webserver) > P_5(Webserver)$$

Wir verwenden Funktionen sowohl für Feature-Attribute als auch für Aggregationsfunktionen, die wir im Folgenden näher erläutern.

3.2.2 Feature-Attribute

Die Feature-Attribute selbst definieren wir in der vorgestellten Syntax als Funktionen. Sie arbeiten auf einem Feature und bekommen dementsprechend als Parameter ein Feature übergeben. Die Funktionen haben den gleichen Namen wie das Feature-Attribut, welches sie auswerten. Für das Feature-Attribut *Latenz* existiert somit auch die einstellige Funktion *Latenz*. Wir können somit die Werte Feature-Attribute

der einzelnen Features miteinander vergleichen. Damit ist unter Anderem folgender Vergleich möglich:

$$Latenz(Datenbank) < Latenz(Dateisystem)$$

Da die Funktionen nur auf Features arbeiten können, die auch das auszuwertende Feature-Attribut bereitstellen, stellen wir die Bedingung, dass die Funktionen nur auf Features mit dem Feature-Attribute angewendet werden. Da die Anwendung auf Features ohne das Attribut nicht zielführend ist, wird dieser Fall nicht betrachtet. Bei der Implementation der Syntax ist zu beachten, dass das Feature das Attribut bereitstellt. Gegebenenfalls ist ein Fehler anzuzeigen.

Nehmen wir nun an, dass beim Konfigurieren für unseren Webserver die Features *Webserver*, *Generierung*, *Protokoll*, *Datenbank* und *MySQL* ausgewählt wurden. Soll nun die Summe aller Latenzen für die Konfiguration einen bestimmten Wert nicht übersteigen, muss der Nutzer für jede Konfiguration ein neues Constraint definieren, damit die Summe korrekt ausgewertet wird. So muss er für jedes Feature mit dem Attribut *Latenz* einen Operanden in das Constraint übernehmen, während für jedes nicht-ausgewählte Feature dieser entfernt oder weggelassen werden muss.

Beispiel. Im folgenden Beispiel sollen die aufsummierten Latenzen der ausgewählten Features kleiner als 30ms sein.

- Wählt der Nutzer die Features *Webserver*, *Protokoll*, *HTTP* und *Generierung*, enthält nur das Feature *Webserver* das Feature-Attribut Latenz. Damit definiert der Nutzer die Anforderung als das Constraint

$$Latenz(Webserver) < 30$$

- Wählt der Nutzer dagegen *Webserver*, *Protokoll*, *HTTP*, *Scriptsprachen*, *PHP* und *Generierung*, würde das von ihm zu definierende Constraint lauten

$$Latenz(Webserver) + Latenz(Scriptsprachen) < 30$$

- Wählt der Nutzer zusätzlich zum letzten Beispiel noch das Feature *SSL* aus, muss er die Formel wie folg erweitern

$$Latenz(Webserver) + Latenz(Scriptsprachen) + Latenz(SSL) < 30$$

Diese Beispiele zeigen, dass der Nutzer für jede einzelne Konfiguration für jedes Attribut die Configuration-Constraints anpassen muss, um mit seinem Constraint den ausgewählten Features in der Konfiguration zu entsprechen. Um dafür den Aufwand für die Nutzer zu reduzieren, verwenden wir Aggregationsfunktionen.

3.2.3 Aggregationsfunktionen

Die Auswertung der Feature-Attribute erfolgt mittels Aggregationsfunktionen, welche wir in unserer Syntax auch als Funktionen definieren. Aggregationsfunktionen nehmen dabei ein Feature-Attribut entgegen und liefern einen Wert im Wertebereich des Feature-Attributs zurück. Da wir als Parameter nur ein Feature-Attribut zulassen, unterstützen wir als Aggregationen nur mathematische Operationen, bei denen die Reihenfolge der Operanden keine Auswirkung auf das Ergebnis hat. Für unsere Betrachtungen wählen wir daher folgende Funktionen.

Um **Summen** über dem Feature-Attribut a zu bilden, verwenden wir die Aggregationsfunktion $sum(a)$. Dabei werden die Werte für a von allen Features, die in der Konfiguration gewählt sind, aufsummiert und das Ergebnis zurückgeliefert.

Um **Produkte** über dem Feature-Attribut a zu bilden, verwenden wir die Aggregationsfunktion $mul(a)$. Alle Werte für a von allen Features in der Konfiguration werden dabei multipliziert und das Ergebnis zurückgeliefert.

Um das **Minimum** und das **Maximum** des Feature-Attributs a auszuwerten, verwenden wir die Aggregationsfunktionen $min(a)$ beziehungsweise $max(a)$. Für alle Features in der Konfiguration wird überprüft, welcher Wert des Attributs a am kleinsten beziehungsweise größten ist, und dieser Wert zurückgegeben.

Um das **arithmetische Mittel** über dem Feature-Attribut a auszuwerten, verwenden wir die Aggregationsfunktion $avg(a)$. Die Werte des Feature-Attributs a von den ausgewählte Featuren in der Konfiguration werden aufsummiert und durch die Anzahl der ausgewählten Features dividiert. Das Ergebnis wird zurückgegeben.

Da sowohl die Division als auch die Subtraktion das Assoziativgesetz nicht mit unterstützen und bei den von uns gewählten Operationen dieses eingehalten werden soll, unterstützen wir diese Aggregationen nicht. Da sich eine Subtraktion jedoch als Summe darstellen lässt, ist diese Operation trotzdem unterstützt. Da wir bei unserer Recherche kein Beispiel für die Division finden konnten, unterstützen wir diese Operation jedoch nicht.

Für den Webserver können wir beispielsweise folgende Constraints bilden, in denen die Feature-Attribute mittels Aggregationsfunktionen ausgewertet werden:

- Aufsummiert sollen die Latenzen einen Wert von 100ms nicht übersteigen:

$$sum(Latenz) < 100$$

- Die Variante der SPL soll zu 98 Prozent der Zeit laufen. Die Ausfallwahrscheinlichkeiten der Features werden dabei multipliziert und dürfen maximal 2 Prozent betragen.

$$mul(Ausfall) \leq 0.02$$

- Es sollen mindestens 1000 Anfragen pro Minute unterstützt werden, wobei das Feature mit der geringsten Anfragenanzahl die Kapazitäten der Variante definiert:

Aggregationsfunktion	**Standardwert d (allgemein)**	**d für ganze Zahlen**
Summe	0	0
Produkt	1	1
Minimum	Obere Datentypgrenze	∞
Maximum	Untere Datentypgrenze	$-\infty$
Durchschnitt	0	0

Tabelle 3.2: Standardwerte für Aggregationsfunktionen über numerischen Feature-Attributen

$$min(MAnf) \geq 1000$$

Damit die Aggregationsfunktionen ein korrektes Ergebnis liefern, müssen wir einige Spezialfälle berücksichtigen. So kann ein ausgewähltes Feature beispielsweise kein Feature-Attribut *Latenz* aufweisen, wie es in unserem Webserver bei den Datenbankmanagementsystemen *MySQL* und *PostgreSQL* der Fall ist. Damit vor der Auswertung der Formel nicht zusätzliche Überprüfungen erfolgen müssen, bietet es sich an, diesen Features bei der Aggregation Standardwerte zuzuweisen, die das Ergebnis der Aggregationsfunktion nicht beeinflussen. Wir verwenden dafür das neutrale Element der jeweiligen Operation. Für die oben genannten Aggregationsfunktionen gelten für numerische Datentypen damit die Standardwerte d aus Tabelle 3.2.

- Summe: Für die Summe ist das neutrale Element die Zahl 0. Addieren wir zu einer Zahl x den Wert 0, ist das Ergebnis x: $x + 0 = x$.
- Produkt: Für ein Produkt verwenden wir das neutrale Element 1. Multiplizieren wir eine Zahl x mit dem Wert 1, erhalten wir die Zahl x: $x \times 1 = x$.
- Minimum: Um das Minimum zu berechnen, verwenden wir als Standardwert die obere Schranke des verwendeten Attributdatentyps. Für die reellen oder natürlichen Zahlen wäre dies ∞, da das Minimum von x und ∞ die Zahl x ist, wenn $x < \infty$ oder ∞, wenn $x = \infty$: $Min(x, \infty) = x$.
- Maximum: Um das Minimum zu berechnen, verwenden wir als Standardwert die untere Schranke des verwendeten Attributdatentyps. Für die reellen oder natürlichen Zahlen wäre dies $-\infty$, da das Maximum von x und $-\infty$ die Zahl x ist, wenn $x > -\infty$ oder $-\infty$, wenn $x = -\infty$: $Max(x, -\infty) = x$.
- Durchschnitt: Da wir beim Durchschnitt Summen durch die Anzahl der Elemente teilen, verwenden wir in den Summen das neutrale Element für die Summe.

Auf Basis der Standardwerte ist es nun möglich, für jede Aggregationsfunktion ein Vorgehen zu definieren, welches unabhängig vom Vorhandensein des Feature-Attributs auf jedem Feature-Attribut im Modell funktioniert. Wir verwenden den Standardwert d als Wert, wenn das Feature-Attribut $a(f)$ für ein Feature f nicht definiert ist oder das Feature nicht ausgewählt ist. Ansonsten verwenden wir den in der Aggregation zugewiesenen konstanten Wert.

Beispiel. In unserer Konfiguration sind die Features *Webserver*, *Generierung*, *Protokoll*, *Datenbank* und *MySQL* ausgewählt. Für *Generierung*, *Protokoll* und *MySQL*

ist dabei das Feature-Attribut *Latenz* nicht definiert. Um die Summe der Latenz zu berechnen, wird daher die Formel

$$sum(Latenz) = Latenz(Webserver) + 0 + 0 + Latenz(Datenbank) + 0$$

gebildet und berechnet.

3.3 Anwendungsbeispiel

Wie in Unterabschnitt 3.1.5 beschrieben, definiert der Nutzer während des Konfigurierens eigene Constraints, um spezielle Anforderungen an die SPL zu formulieren, die nicht durch das Feature-Modell vorgegeben werden. Diese Constraints sind dabei ähnlich aufgebaut wie die im Feature-Modell hinterlegten Constraints, werden jedoch nach dem Syntax aus Abschnitt 3.2 um Feature-Attribute und Aggregationsfunktionen erweitert.

Um die Definition der Configuration-Constraints in den Prozess des Konfigurierens einzubinden, gibt es zwei Möglichkeiten. Zum einen kann die Definition der Constraints vor der Auswahl der Features erfolgen, zum anderen kann eine Definition der Constraints auch während des gesamten Prozesses erlaubt werden. Wird diese Funktionalität an den Anfang des Prozesses gestellt, wird damit eine Trennung der einzelnen Schritte für die Spezifizierung des speziellen Anwendungsfall des Nutzers und die Schritte für die Feature-Auswahl vorgenommen. Nutzer haben nicht die Möglichkeit, während des Auswählen der Features Anpassungen in den Constraints vorzunehmen. Da etwaige Fehler in den Constraints damit eine erneute Feature-Auswahl erfordern, müssen dem Nutzer die gesamten Anforderungen bereits zum Beginn des Konfigurierens klar definiert sein.
Wird auf diese strikte Trennung verzichtet und die Funktion im gesamten Konfigurationsprozess erlaubt, ist eine Anpassung der Constraints dagegen möglich. Fehler in den Constraints, beispielsweise eine zu niedrig gesetzte Schranke für die Latenz, können in diesem Fall im Nachhinein korrigiert werden. Vor allem ist durch den Nutzer direkt erkennbar, welche Auswirkungen die Anpassung einzelner Constraints hat.

Werden automatisiert die Analysen aus Abschnitt 2.4 durchgeführt, so kann der Nutzer beispielsweise eine Veränderung der Anzahl der möglichen Konfigurationen sehen. Die zusätzlich definierten Constraints werden dabei durch Tools wie Solver zusammen mit dem Feature-Modell und Cross-Tree-Constraints ausgewertet. Dafür werden die Configuration-Constraints in aussagenlogische Formeln umgewandelt. Für die Solver ist es dabei irrelevant, zu welchem Zeitpunkt im Konfigurationsprozess sie die Kombination aus Configuration-Constraints und Feature-Modell lösen. Sie behandeln diese Kombination als eine einzige Konjunktion von Formeln, die es zu lösen gilt. Sie unterstützen also sowohl die Trennung von Definition der Configuration-Constraints von der Feature-Auswahl als auch die Integration der Constraint-Definition in den gesamten Prozess. Da der zweite Fall somit auch bei den von uns gewählten SMT-Solvern unterstützt ist und eine Verallgemeinerung des ersten darstellt, untersuchen wir in der Arbeit den zweiten Fall.

Nehmen wir an ein Nutzer möchte einen Webserver mit der SPL aus Abbildung 2.2 verwenden. Er habe dabei folgende Anforderungen:

- Eine Antwort muss innerhalb von 100ms erfolgen
- Der Webserver soll mindestens 700 Nutzer pro Minute unterstützen
- Der Webserver darf maximal eine Ausfallrate von 1 Prozent haben

Da wir keine Freitextform für die Anfragen unterstützen, müssen diese Anforderungen entsprechend der Syntax aus Abschnitt 3.2 definiert werden. Daher formuliert der Nutzer diese um:

- Durch Domänenwissen weiß der Nutzer, dass die Latenzen sich addieren. Wenn eine Softwarekomponente 20ms für die Bearbeitung einer Anfrage benötigt, und das Ergebnis dieser Bearbeitung durch eine weitere Softwarekomponente in 10ms in ein anderes Format umgewandelt wird, ist die finale Bearbeitungszeit 30ms. Der Nutzer wählt damit die Aggregationsfunktion *sum* für das Aggregieren der Latenzen. Er definiert die Anforderung als $sum(Latenz) < 100$.
- Die SPL für den Webserver stellt kein Feature-Attribut bereit, welches die Nutzer pro Minute beschreibt. Stattdessen gibt es das Feature-Attribut *MAnf*, welches die maximalen Anfragen pro Minute für ein Feature erfasst. Dies entspricht nicht eins zu eins der Anforderung für den Nutzer, das Feature-Attribut kann jedoch für die Beschreibung der Anforderung in unsere Syntax verwendet werden. Um auf der sicheren Seite zu sein, wählt der Nutzer noch einen Puffer zu seiner Schranke von 700 Nutzern und verzwanzigfacht diesen Wert auf 14000. Theoretisch wäre es nun 700 Nutzern möglich 200 Anfragen pro Minute zu stellen. Die Anforderung lautet in unserer Syntax nun $min(MAnf) > 14000$.
- Für die Ausfallrate ist die Anforderung als Maximum definiert und die Anforderung wird durch den Nutzer als $max(Ausfall) < 1$ definiert.

Die Anforderungen des Nutzer fasst dieser in folgender Formel zusammen:

$$sum(Latenz) < 100 \land min(MAnf) > 14000 \land max(Ausfall) < 1$$

Anhand des Beispiels lassen sich verschiedene Faktoren ermitteln, die für die Anwendung der Configuration-Constraints mit dem Syntax aus Abschnitt 3.2 zu beachten sind.

- Für die korrekte Erstellung der Constraints ist es wichtig, dass der Ersteller wissen über die Domäne der SPL besitzt. So basiert die Auswahl der Aggregationsfunktionen auf den Eigenschaften der vorhandenen Feature-Attribute.
- Die von uns beschriebene Syntax beschränkt sich auf eine Auswahl an Aggregationsfunktionen. Abhängig vom Anwendungsfall der Nutzer bei der Konfiguration kann diese Auswahl nicht ausreichen.
- Anforderungen des Nutzers bei der Konfiguration können je nach Anwendungsfall nicht durch Configuration-Constraints ausgedrückt werden, wenn die Attribute die Anforderungen nicht unterstützen.

Da der Nutzer die Constraints nun definiert hat, beginnt der Auswahlprozess der Features. So werden erforderliche Features automatisch durch Decision Propagation ausgewählt. Dies kann bereits dazu führen, dass aufgrund der zusätzlichen Configuration-Constraints keine valide Konfiguration existiert, ist in unserem Beispiel jedoch nicht der Fall. Für den Nutzer werden automatisch die Features *Webserver*, *Protokoll*, *File*, *Generierung*, *Statisch* und *Dateisystem* als ausgewählte Features in die Konfiguration übernommen. Im Rahmen der Decision Propagation wird ebenfalls ermittelt, ob es Features gibt, die für die derzeitige Auswahl nicht mehr zulässig sind. In unserem Fall ist bereits von Beginn an bekannt, dass die Features *PostgreSQL* und *WAR* nicht mehr zulässig sind, da sie die Anforderung nach der Ausfallsicherheit verletzen. Der Nutzer wählt nun das Feature *HTTP*, wodurch die Features *File*, *Generierung*, *Statisch* und *Dateisystem* keine Core-Features mehr sind. Dies wird durch Decision Propagation überprüft. Da der Nutzer mit der Auswahl zufrieden ist, generiert er die Variante der SPL und hat den Konfigurationsprozess damit abgeschlossen.

3.4 Abbildung auf SMT-Syntax

Für die Durchführung der Analysen verwenden wir SMT-Solver, da SAT-Solver nicht den für die Syntax aus Abschnitt 3.2 erforderlichen Umfang an Funktionalitäten unterstützen (vgl. Abschnitt 2.4). Um die Constraints aus Abschnitt 3.2 bei den Analysen auswerten zu können, müssen die Constraints im Syntax des SMT-Solvers definiert werden (vgl. Barrett et al. [2015]). Je Feature-Attribut werden weitere Formeln ermittelt: die Zuweisung des Wertes des Feature-Attributs und die Aggregation über dem Attribut. Alle Formeln werden den Constraints im Feature-Modell mittels Konjunktionen angehängt.

Zuweisung des Attributwertes. Wie in Abschnitt 3.2 beschrieben, muss einem Feature-Attribut $a(f)$ für ein Feature f der konstante Wert aus dem Feature-Modell zugewiesen werden, wenn dieses Feature ausgewählt wird und für $a(f)$ ein konstanter Wert definiert wurde. Wenn k für den konstanten Wert aus dem Feature-Modell steht, bilden wir für die Wertzuweisung die folgende Formel.

$$f \Rightarrow (a(f) = k) \tag{3.1}$$

Da wir die Auswertung der Feature-Attribute innerhalb von Aggregationsfunktionen durchführen, können wir auf den Standardwert d für die jeweilige Aggregation zugreifen. Ist daher ein Feature-Attribut für ein Feature nicht definiert, verwenden wir statt k den Standardwert der verwendeten Aggregation.

Der Standardwert erlaubt es auch, dass bei der Auswertung eines Feature-Attributs die nicht ausgewählten Features berücksichtigt werden. Wir weisen in diesem Fall dem Feature-Attribut den Standardwert der Aggregationsfunktion zu. Wenn der Standardwert d sei, dann wird zur Formel aus Gleichung 3.1 auch folgende Formel hinzugefügt, um den Fall abzubilden, dass ein Feature nicht ausgewählt ist.

$$\neg f \Rightarrow (a(f) = d) \tag{3.2}$$

Beispiel. Wenn die Summe der Latenzen für eine gegebene Konfiguration ermittelt werden soll, werden für alle Features die gerade genannten Formeln gebildet. Nehmen

wir an, die ausgewählte Konfiguration bestünde aus den Features *Webserver*, *Protokoll*, *HTTPs*, *SSL* und *Generierung*. Der Standardwert d für die Summe beträgt 0. Für alle Features erstellen wir die Formeln analog zu Gleichung 3.1 und Gleichung 3.2. Für das Feature *Webserver* erhalten wir damit die folgenden Formeln.

$$\begin{aligned} Webserver &\Rightarrow (Latenz(Webserver) = 20) \\ \neg Webserver &\Rightarrow (Latenz(Webserver) = 0) \end{aligned}$$

Da Webserver ausgewählt ist, wird die erste Formel durch den Solver verwendet, da *Webserver* als wahre Aussage angesehen wird.
Das Feature *Protokoll* besitzt kein Feature-Attribut *Latenz*. Daher werden folgende Formeln für das Feature gebildet.

$$\begin{aligned} Protokoll &\Rightarrow (Latenz(Protokoll) = 0) \\ \neg Protokoll &\Rightarrow (Latenz(Protokoll) = 0) \end{aligned}$$

Um die Werte für die Feature-Attribute zu den Features zu speichern, verwenden wir in den Constraints für den SMT-Solver Variablen. Diese Variablen bestehen dabei aus dem Namen des Features, zum Beispiel *Webserver*, und dem Namen des Feature-Attributs, beispielsweise *Latenz*. Diese Variablen lassen sich entweder direkt bei der Modellerstellung definieren, oder nach Bedarf beim Auswerten der Configuration-Constraints. Um das Vorgehen zu vereinfachen und das Worst-Case-Szenario mit den meisten Variablen zu untersuchen, definieren wir die Variablen bereits bei der Erstellung des Modells. Dabei wird für jedes Feature für jedes Feature-Attribut eine Variable in der Form $Latenz(Webserver)$ angelegt. Für die Latenz des Webservers und alle anderen Variablen erhalten wir damit jeweils die zwei Formeln.

$$\begin{aligned} Webserver &\Rightarrow (Latenz(Webserver) = k) \\ \neg Webserver &\Rightarrow (Latenz(Webserver) = d) \end{aligned}$$

Bilden von Aggregationen über einem Feature-Attribut. Entsprechend der Syntax aus Abschnitt 3.2 sei a das auszuwertende Feature-Attribut. Es dient als Parameter für jede in Unterabschnitt 3.2.3 definierte Aggregationsfunktion. Bei der Definition der Aggregationfunktionen ist dabei anhand der Definition der SMTLib zu beachten, dass das Ergebnis der Aggregationsfunktion den gleichen Wertebereich aufweist wie die Werte des Feature-Attributs a (vgl. [Barrett et al., 2015]).

Eine **Summe** über dem Feature-Attribut a wird gebildet, indem die Werte für die jeweiligen Feature-Attribute im Feature-Modell aufsummiert werden. Dies wird in folgender Gleichung für die Features f_1 bis f_n realisiert.

$$sum(a) = a(f_1) + a(f_2) + \cdots + a(f_n) \tag{3.3}$$

Analog wird zur Summe wird das **Produkt** über dem Feature-Attribut a gebildet. Hierbei werden die Werte für die jeweiligen Features miteinander multipliziert.

$$mul(a) = a(f_1) \times a(f_2) \times \cdots \times a(f_n) \tag{3.4}$$

Um das **Minimum** beziehungsweise **Maximum** über dem Feature-Attribut a abzubilden, müssen paarweise Vergleiche der Feature-Attributwerte durchgeführt werden. Jeder Vergleich resultiert dabei in einer neuen Variable.
Zwei beliebige unterschiedliche Features f_i und f_j werden miteinander verglichen, um das Minimum des Feature-Attributs a für die Konfiguration zu ermitteln. Als Ergebnis des Vergleichs der beiden Features erhalten wir eine Variable min_{ij}, welche entweder den Wert $a(f_i)$ oder den Wert $a(f_j)$ zugewiesen bekommt.

$$\begin{aligned} (a(f_i) < a(f_j)) &\Rightarrow (min_{ij} = a(f_i)) \\ \neg(a(f_i) < a(f_j)) &\Rightarrow (min_{ij} = a(f_j)) \end{aligned} \tag{3.5}$$

Für weitere Features f_k wird nun solange das Minimum min_{ijk} von min_{ij} und $a(f_k)$ gebildet, bis alle Features im Minimum min_{ijk} berücksichtigt werden.

$$\begin{aligned} (min_{ij} < a(f_k)) &\Rightarrow (min_{ijk} = min_{ij}) \\ \neg(min_{ij} < a(f_k)) &\Rightarrow (min_{ijk} = a(f_k)) \end{aligned} \tag{3.6}$$

Die letzte Variable in der Iteration ist damit das Minimum $min(a)$ über dem Feature-Attribut a.
Analog zu Gleichung 3.5 und Gleichung 3.6 erfolgt das Vorgehen für das Maximum über dem Featureattribut a:

$$\begin{aligned} (a(f_i) > a(f_j)) &\Rightarrow (max_{ij} = a(f_i)) \\ \neg(a(f_i) > a(f_j)) &\Rightarrow (max_{ij} = a(f_j)) \\ (max_{ij} > a(f_k)) &\Rightarrow (max_{ijk} = max_{ij}) \\ \neg(max_{ij} > a(f_k)) &\Rightarrow (max_{ijk} = a(f_k)) \end{aligned} \tag{3.7}$$

Das Ergebnis der letzten Iteration ist das Maximum $max(a)$ über dem Feature-Attribut a.

Um das arithmethische Mittel über dem Feature-Attribut a zu ermitteln, benötigen wir die **Anzahl der ausgewählten Features**. Um diese zu ermitteln, gehen wir iterativ vor: Ausgehend vom Wurzel-Feature iterieren wir über alle Features im Feature-Modell. Wenn die Variable für das Feature wahr ist, erhöhen wir die Anzahl um 1. Dies wiederholen wir, solange es noch nicht behandelte Features gibt. Das Vorgehen wird in Gleichung 3.8 und Gleichung 3.9 verdeutlicht. Für das Wurzel-Feature f_1 berechnen wir die Variable $count_1$.

$$\begin{aligned} f_1 &\Rightarrow (count_1 = 1) \\ \neg f_1 &\Rightarrow (count_1 = 0) \end{aligned} \tag{3.8}$$

Für weitere Features f_j addieren wir 1 zur vorherigen Variable.

$$\begin{aligned} f_j &\Rightarrow (count_j = count_{j-1} + 1) \\ \neg f_j &\Rightarrow (count_j = count_{j-1}) \end{aligned} \tag{3.9}$$

Das Ergebnis der letzten Iteration ist die Anzahl der in der Konfiguration enthaltenen Features $count$. Wird das Vorgehen für n Features durchgeführt, erhalten wir das Ergebnis in der Variable $count_n$, welches wir der Variable $count$ zuweisen.

$$count = count_n.$$

Um das **arithmetische Mittel** über dem Feature-Attribut a zu bilden, greifen wir auf die Anzahl der in der Konfiguration enthaltenen Features *count* und die Berechnung der Summe über dem Attribut $sum(a)$ zurück.

$$avg(a) = sum(a)/count \tag{3.10}$$

Kombination verschiedener Aggregationsfunktionen. Tritt der Fall ein, dass ein oder mehrere Constraints definiert werden, die unterschiedliche Aggregationsfunktionen verwenden, werden durch den SMT-Solver fehlerhafte Ergebnisse ermittelt. Die Aggregationsfunktion Produkt verwendet einen anderen Standardwert als die Funktion arithmetisches Mittel. Werden diese jeweils der gleichen Variable zugewiesen, ist es abhängig von der Reihenfolge der Ausführung der Zuweisungen, welcher Standardwert bei den Berechnungen verwendet wird.
Auf Basis der Ausfallwahrscheinlichkeit sollen beispielsweise durchschnittliche Wahrscheinlichkeit, mit der ein ausgewähltes Feature der SPL ausfällt, und die Wahrscheinlichkeit, dass die gesamte SPL ausfällt, berechnet werden.

- Um die Wahrscheinlichkeit zu berechnen, dass eines der ausgewählten Features ausfällt, wird das arithmetische Mittel über dem Feature-Attribut *Ausfall* berechnet.

$$avg(Ausfall)$$

- Um die Wahrscheinlichkeit zu berechnen, dass die gesamte SPL ausfällt, werden die Ausfallwahrscheinlichkeiten multipliziert.

$$mul(Ausfall)$$

Für das Feature *MySQL* greifen wir dabei auf die Variable *MySQL.Ausfall* zurück, die entweder den Wert aus dem Feature-Modell oder den Standardwert der Aggregation zugewiesen bekommt. Nach dem bisherigen Vorgehen definieren wir dabei die folgenden vier Formeln für die Wertzuweisung.

- $MySQL \Rightarrow (Ausfall(MySQL) = 1)$
- $\neg MySQL \Rightarrow (Ausfall(MySQL) = 1)$
- $MySQL \Rightarrow (Ausfall(MySQL) = 1)$
- $\neg MySQL \Rightarrow (Ausfall(MySQL) = 0)$

Dabei kommt es bei der zweiten und vierten Formel zu einem Konflikt, da die Variable zweimal mit einem anderen Wert belegt wird. Um diesen Fall zu vermeiden, fügen wir an jede Variable, die für die Wertzuweisung erstellt wird, den Kontext an, indem sie verwendet wird. Dadurch erhalten wir für das Feature *MySQL* für das Feature-Attribut *Ausfall* die Variablen aus Tabelle 3.3. Die gerade genannten Formeln werden damit wie folgt umgewandelt:

Aggregationsfunktion	**Variablenname**
Summe	$Ausfall().sum$
Produkt	$Ausfall(MySQL).mul$
Minimum	$Ausfall(MySQL).min$
Maximum	$Ausfall(MySQL).max$
Durchschnitt	$Ausfall(MySQL).avg$

Tabelle 3.3: Variablennamen für das Feature *MySQL* für das Feature-Attribut *Ausfall*

- $MySQL \Rightarrow (Ausfall(MySQL).mul = 1)$
- $\neg MySQL \Rightarrow (Ausfall(MySQL).mul = 1)$
- $MySQL \Rightarrow (Ausfall(MySQL).avg = 1)$
- $\neg MySQL \Rightarrow (Ausfall(MySQL).avg = 0)$

Vorgehen. Um die vom Nutzer definierten Constraints für die SMT-Solver bereitzustellen, müssen daher folgende Schritte durchgeführt werden.

- Ist das Modell erstellt oder wird es geladen, werden alle notwendigen Variablen für die Constraints im SMT-Syntax definiert. Für jedes Feature wird für jedes Feature-Attribut und jede Aggregationsfunktion jeweils eine Variable mit dem Namen *<Feature-Attribut>*(*<Feature>*).*<Aggregationsfunktion>* erstellt.
- Die Configuration-Constraints werden in Constraints in SMT-Syntax umgewandelt, wobei die bereits definierten Variablen verwendet werden.
- Alle Constraints in SMT-Syntax werden durch Konjunktionen miteinander verknüpft und mit den Formeln aus dem Feature-Modell an den Solver übergeben.

4. Implementierung

Um das Konzept aus Kapitel 3 zu evaluieren, entwickeln wir mithilfe eines Java-Frameworks auf Basis von JavaSMT[1] einen Prototypen[2]. Durch das Framework werden zusätzliche Wrapper- und Utility-Klassen bereitgestellt, die unter anderem verschiedene Formate von Feature-Modellen unterstützen. So wird auch das von FeatureIDE[3] definierte XML-Format unterstützt.

4.1 Umwandeln des Feature-Modells

In Abbildung 4.1 wird der Algorithmus für das Umwandeln eines Feature-Modells im XML-Format grob dargestellt. Aus einem erweiterten Feature-Modell werden die Features, Modell-Struktur, Cross-Tree-Constraints und Feature-Attribute gelesen. Auf Basis der Features und Feature-Attributen werden Variablen definiert, die später in den Aggregationsfunktionen ausgewertet werden sollen. Sind die Variablen definiert, werden Formeln für die Einschränkung der Werte für diese Variablen erstellt. Diese werden im letzten Schritt mit den anderen Formeln aus dem Feature-Modell konjugiert und als Ergebnis des Prozesses erhält man eine einzige Formel mit allen Constraints.

4.1.1 Erweitertes Feature-Modell

Dem Algorithmus wird als Input eine XML-Struktur vom Feature-Modell im Format von FeatureIDE übergeben. Diese Struktur wird innerhalb von FeatureIDE verwendet, um die Ergebnisse des Modellierens des Feature-Modells zu speichern. Sie enthält das erweiterte Feature-Modell mit allen Features, Feature-Attributen und Cross-Tree-Constraints. Das komplette Feature-Modell aus Abbildung 2.2 im eben erwähnten XML-Format ist im Anhang in Quelltext A.1 dargestellt.

Die XML-Struktur für das Feature-Modell ist dabei in zwei Teile unterteilt. Zum einen ist unter *struct* die Struktur des Feature-Modells erfasst. Hierbei ist die Baumstruktur

[1] https://github.com/skrieter/spldev
[2] https://github.com/dhohmann/formula-analysis-javasmt bzw. https://github.com/dhohmann/formula
[3] https://github.com/FeatureIDE/FeatureIDE

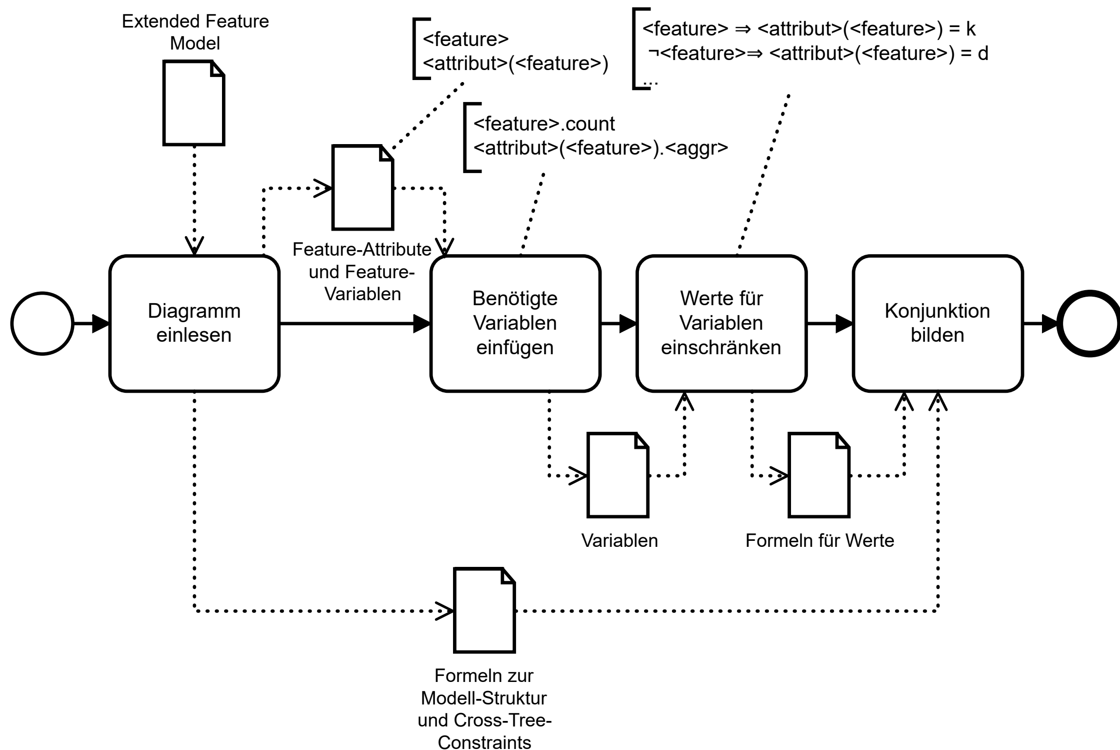

Abbildung 4.1: Umwandeln des Feature-Modells in eine aussagenlogische Formel

aus Abbildung 2.2 mit den Eltern-Kind-Abhängigkeiten der Features nachempfunden. So ist das Wurzelfeature *Webserver*, wie in Quelltext 4.1 dargestellt, der Beginn der XML-Struktur für das Feature-Modell, welches alle untergeordneten Features mittels einer Und-Verknüpfung miteinander verknüpft. Da das Feature zudem ein Kern-Feature ist, ist es als *mandatory* gekennzeichnet.

```
<and mandatory="true" name="Webserver">
    <!-- ... -->
</and>
```

Quelltext 4.1: Wurzel-Feature aus Abbildung 2.2 als XML

Zum anderen enthält die XML-Struktur die Cross-Tree-Constraints unter *constraints*. Dabei ist jede einzelne Bedingung als *rule*-Tag definiert. In dieser wird eine aussagenlogische Formel dargestellt. Das Constraint $\neg(HTTP \vee HTTPS) \Rightarrow File$ aus dem Feature-Modell aus Abbildung 2.2 wird dabei wie folgt in Quelltext 4.2 dargestellt.

```
<rule>
    <imp>
        <not>
            <disj>
                <var>HTTP</var>
                <var>HTTPS</var>
            </disj>
        </not>
        <var>File</var>
```

```
    </imp>
</rule>
```

Quelltext 4.2: Beispielhaftes Cross-Tree-Constraint als XML

Für die Umwandlung eines Feature-Modells ohne Feature-Attribute in eine aussagenlogische Formel stellt das verwendete Java-Framework bereits die gesamte Funktionalität für das Einlesen einer XML-Datei im Feature-IDE-Format bereit. Um auch erweiterte Feature-Modelle mit Feature-Attributen zu unterstützen, erweitern wir den bestehenden Prozess insofern, dass beim Einlesen eines Features die Feature-Attribute pro Feature inklusive ihres Wertes in einer zusätzlichen Datenstruktur gespeichert werden.

4.1.2 Feature-Attribute

Feature-Attribute im XML-Format von FeatureIDE werden mittels `attribute`-Tags gekennzeichnet. Dieses Tag enthält dabei mehrere XML-Attribute, die die Daten zum Feature-Attribut enthalten. Für den *Webserver* aus Abbildung 2.2 wird das Feature-Attribut *Latenz* in Quelltext 4.3 dargestellt.

```
<and mandatory="true" name="Webserver">
    <!-- ... -->
    <attribute name="Latenz" type="long" unit="ms" value="10"/>
</and>
```

Quelltext 4.3: Feature-Attribut *Latenz* vom Webserver als XML

Definition im XML. Das Feature-Attribut besitzt einen Namen, festgehalten mit dem XML-Attribut `name`. Dieser identifiziert für jedes Feature das Feature-Attribut. Daher ist es ausgeschlossen, dass für ein Feature mehrere `attribut`-Tags mit dem gleichen Namensattribut definiert sind.
Das Feature-Attribut besitzt einen Datentyp, der den Wertebereich einschränkt. Dieser Datentyp wird über das XML-Attribut `type` definiert. In Quelltext 4.3 ist für die *Latenz* des *Webserver*s werdem dem Feature-Attribut nur ganze Zahlen zugewiesen, da der Wertebereich `long` ist.
Die Werte für ein Feature-Attribut werden mit dem XML-Attribut `value` festgehalten. Im Beispiel ist dies der Wert `10`. Die Einheit dieses Wertes wird im XML-Attribut `unit` festgehalten. Da wir mit unserem Prototypen das Konzept überprüfen vollen, beachten wir die Einheit bei unserer Implementierung nicht.

Erweiterung des Einlesens. Der vom Java-Framework bereitgestellte Algorithmus unterstützt noch nicht das Einlesen von Feature-Attributen. Um die Feature-Attribute einzulesen, erweitern wir daher die Umwandlung von Features. Für jedes Feature-Element im XML überprüfen wir, ob es in den Kind-Knoten Knoten vom Typ `attribute` gibt. Ist dies der Fall, lesen wir die Informationen aus den entsprechenden `attribute`-Knoten und speichern diese in einer separaten Map. Dabei verwenden wir das in Abschnitt 3.2 definierte Format *<attribute>(<feature>)* für den Schlüssel. Aus dem XML aus Quelltext 4.3 generieren wir daher für das Feature *Webserver* einen Schlüssel *Latenz(Webserver)* und speichern den Ganzzahlwert 10 unter diesem Schlüssel ab.

4.1.3 Variablen einfügen

Sind alle Informationen aus dem Feature-Modell ausgelesen, legen wir alle benötigten Variablen an, die keine Feature repräsentieren. Dazu gehören

4.1.3.1 Hilfsvariable für das arithmetische Mittel

Wie in Unterabschnitt 3.2.3 beschrieben, wird zum Berechnen des arithmetischen Mittels die Anzahl der ausgewählten Features benötigt. Um diese bereitzustellen, führen wir beim Umwandeln des Feature-Modells von einem XML-Format in eine aussagenlogische Formel Hilfsvariablen ein.

Die Variable *count* repräsentiert die Gesamtanzahl an ausgewählten Features. Sie wird gleich der Summe aller Variablen gesetzt, die die Form $<feature>.count$ aufweisen. Dafür definieren wir nach dem Einlesen aller Features für jedes Feature die entsprechende Variable. Zusätzlich definieren wir zwei Formeln:

- Wenn das Feature, für welches wir die Variable *feature.count* anlegen, in der Konfiguration später ausgewählt ist, soll die Variable den Wert 1 aufweisen. Dafür fügen wir folgende Formel ein:

$$<feature> \Rightarrow <feature>.\text{count} = 1$$

- Wenn das Feature, für welches wir die Variable *feature.count* anlegen, in der Konfiguration später nicht ausgewählt ist, soll sie nicht zum Wert für die Variable *count* beitragen. Da wir in der Formel für *count* eine Summe verwenden, kann dies über das neutrale Element der Summe erreicht werden. Daher definieren wir folgende Formel:

$$\neg <feature> \Rightarrow <feature>.count = 0$$

4.1.3.2 Variablen für Feature-Attribute in Hinblick auf Aggregationsfunktionen

Abhängig von der Aggregationsfunktion, unterscheidet sich der Standardwert, den ein Feature-Attribut zugewiesen bekommt (vgl. Tabelle 3.2). Um mehrere Standardwerte zu unterstützen, führen wir weitere Variablen für die Feature-Attribute ein.

Für ein Feature-Attribut aus dem Feature-Modell führen wir für jede Aggregationfunktion eine neue Variable mit dem gleichen Wertebereich ein. Dieser neuen Variable geben wir dabei den Namen des Feature-Attributs mit dem Namen der jeweiligen Aggregationsfunktion als Suffix. So werden für ein Feature-Attribut $Latenz(Webserver)$ für die Aggregationsfunktionen *sum*, *mul* und *avg* die Variablen $Latenz(Webserver).sum$, $Latenz(Webserver).mul$ und $Latenz(Webserver).avg$ zu den Variablen hinzugefügt. Schränken wir nun die möglichen Werte, die die Variable für eine Aggregationsfunktion annehmen kann, auf den Standardwert der Aggregationsfunktion ein, beeinflusst dies nicht die anderen Variablen. Die initiale Variable, die aus dem Feature-Modell gelesen wird, soll den Wert des Feature-Attributs aus dem Modell enthalten. Für das Beispiel mit der Latenz ist dies $Latenz(Webserver)$. Für das Feature-Attribut *Latenz* für das Feature *Webserver* werden somit folgende Variablen erstellt:

- $Latenz(Webserver)$
- $Latenz(Webserver).sum$
- $Latenz(Webserver).mul$
- $Latenz(Webserver).avg$

4.1.4 Werte für Variablen einschränken

Die Variablen für eine Feature-Attribut-Aggregationsfunktion-Kombination dürfen nur den Standardwert der Aggregationsfunktion und den Wert des Feature-Attributs aus dem Feature-Modell annehmen (vlg. Kapitel 3). Ist ein Feature ausgewählt, müssen die Variablen ein Feature-Attribute zu diesem Feature nur den im Feature-Modell definierten Wert annehmen können. Falls das Feature nicht ausgewählt ist, darf die Variable keine Auswirkungen auf das Ergebnis haben, muss also den Standardwert annehmen. Um dies zu gewährleisten, definieren wir für jede Kombination aus Feature-Attribut und Aggregationsfunktion zwei Formeln.

Wenn das Feature zum Feature-Attribut ausgewählt ist, ist die neue Variable gleich dem eingelesenen Wert aus dem Feature-Attribut. Für das Feature-Attribut *Latenz* vom *Webserver* bedeutet dies, dass folgende Formeln für die Aggregationsfunktion der Summe durch den Solver zu lösen sind:

$$\begin{aligned} Webserver &\Rightarrow (Latenz(Webserver).sum = Latenz(Webserver)) \\ \neg Webserver &\Rightarrow (Latenz(Webserver).sum = 0) \end{aligned}$$

Da wir die Variable $Latenz(Webserver)$ verwenden, um den Attribut-Wert aus dem Feature-Modell zu repräsentieren, müssen wir gewährleisten, dass diese Variable nur genau diesen Wert annehmen kann. Daher definieren wir eine zusätzliche Formel:

$$Latenz(Webserver) = 10$$

Damit sind die für die Configuration-Constraints benötigten Variablen erstellt und können in diesen verwendet werden.

4.2 Anwenden von Configuration-Constraints

Wie in Unterabschnitt 3.1.5 beschrieben, stellen wir mittels Configuration-Constraints für die Nutzer die Möglichkeit bereit, während der Konfiguration zusätzliche Einschränkungen der SPL auf Basis der Feature-Attribute zu treffen. Dafür ist es nötig, dass die Configuration-Constraints durch den Nutzer definiert und durch das Programm analysiert und angewendet werden können.

4.2.1 Format für Configuration-Constraints

Um die Configuration-Constraints durch den Nutzer bereitstellen zu lassen, adaptieren wir das in Abschnitt 4.1 beschriebene Format für die Cross-Tree-Constraints. In diesem Format sind bereits die Booleschen Operatoren $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$ durch entsprechende Schlüsselwörter definiert. Zudem erlaubt es die Verwendung von Features als Variablen mit dem Schlüsselwort `var` und es existieren bereits Parser, um aus den daraus gebildeten XML-Strukturen aussagenlogische Formeln zu bilden.

```
<constraints>
    <rule>
        ...
    </rule>
    <rule>
        ...
    </rule>
    <rule>
        ...
    </rule>
</constraints>
```

Quelltext 4.4: Grundlegene XML-Struktur für Configuration-Constraints

Für die Configuration-Constraints adaptieren wir die Struktur, dass der Wurzelknoten durch einen Knoten vom Typ `constraints` gebildet wird, welcher wiederum Knoten vom Typ `rule` als Kindknoten enthält (vgl. Quelltext 4.4). Im XML-Format für die Cross-Tree-Constraints enthalten die Knoten vom Typ `rule` einen Knoten, der für eine der Booleschen Operatoren steht. Dies adaptieren wir auch für die Configuration-Constraints, erweitern die Syntax jedoch um weitere Operatoren.

Um Vergleiche zweier Terme mittels der Operatoren $<$, $>$, $\leq$, $\geq$ und $=$ zu erlauben, erweitern wir das XML-Format für die Kindknoten des Typs `rule` um die Knoten `lessThan` ($<$), `greaterThan` ($>$), `lessEquals` ($\leq$), `greaterEquals` ($\geq$) und `equals` ($=$). Jeder dieser Knoten erlaubt zwei Kindknoten, die entweder Variablen, Konstanten oder Aggregationsfunktionen repräsentieren.

Für Konstanten erweitern wir das XML-Format um den Knoten `const`, um konstante Werte in Vergleichen zu erlauben. Da in den Logiken der SMTLib zwischen verschiedenen Datentypen unterschieden wird (vgl. Barrett et al. [2015]), ist es dabei erforderlich, dass für jede Konstante der Datentyp für den Wertebereich festgelegt wird. Dies erlauben wir über das XML-Attribut `type`, welches für Ganzzahlen den Wert *long* und für reelle Zahlen den Wert *double* zugewiesen bekommt. In Quelltext 4.5 werden die ganze Zahl 42 und die reelle Zahl 1,6 dargestellt. Der Wert der Konstante wird dabei als Text innerhalb des Knotens angegeben.

```
<const type="long">42</const>
<const type="double">1.6</const>
```

Quelltext 4.5: Beispiel für Konstanten in Configuration-Constraints

Über Aggregationsfunktionen erlauben wir, dass Nutzer auf ausgewertete Feature-Attribute zugreifen können, um das Ergebnis der Auswertungen in Configuration-Constraints in Vergleichen zu verwenden (vgl. Unterabschnitt 3.2.3). Um diese in der XML-Struktur für die Configuration-Constraints darzustellen, erweitern wir das XML-Format um weitere Knoten.

- Über Knoten vom Typ `avg` stellen wir das arithmetische Mittel über einem Feature-Attribut für die Constraints bereit.
- Über Knoten vom Typ `sum` stellen wir die Summe über einem Feature-Attribut für die Constraints bereit.
- Über Knoten vom Typ `prod` stellen wir das Produkt über einem Feature-Attribut für die Constraints bereit.

Für jede Aggregationsfunktion muss dabei angegeben werden, welchen Datentypen die Funktion als Ergebnis zurückliefert. Wie auch bei den Konstanten wird dies über das XML-Attribut `type` angegeben, welches die Werte *long* für ganze Zahlen und *double* für reelle Zahlen annehmen kann. Das auszuwertende Feature-Attribut wird als Text innerhalb des Knotens angegeben. Um die Latenz aus Abbildung 2.2 in den Configuration-Constraints in Vergleichen auszuwerten, können beispielsweise die Aggregationsfunktionen aus Quelltext 4.6 verwendet werden.

```
<avg type="long">Latenz</avg>
<sum type="long">Latenz</sum>
<prod type="double">Latenz</prod>
```

Quelltext 4.6: Beispiel für Aggregationsfunktionen in Configuration-Constraints

Ein Nutzer habe folgende Anforderungen an die SPL aus Abbildung 2.2:

- Eine Antwort muss innerhalb von 100ms erfolgen. sum
- Die Wahrscheinlichkeit, dass alle Features ausfallen, darf nur 1 Prozent betragen.
- Die Wahrscheinlichkeit, dass ein Feature ausfällt, darf nur 5 Prozent betragen.

Diese Anforderungen übertrage der Nutzer jeweils in die Syntax aus Abschnitt 3.2. Er erhält drei zu erfüllende Formeln:

- sum(Latenz)$<$100
- prod(Ausfall)$<$1
- avg(Ausfall)$<$5

Diese Formeln definiere der Nutzer nun in dem XML-Format für Configuration-Constraints. Dadurch erhält er eine XML-Struktur wie in Quelltext 4.7.

```xml
<constraints>
    <rule>
        <lessThan>
            <sum type="long">Latenz</sum>
            <const type="long">100</const>
        </lessThan>
    </rule>
    <rule>
        <lessThan>
            <prod type="double">Ausfall</prod>
            <const type="double">1</const>
        </lessThan>
    </rule>
    <rule>
        <lessThan>
            <prod type="double">Ausfall</prod>
            <const type="double">5</const>
        </lessThan>
    </rule>
</constraints>
```

Quelltext 4.7: Beispiel einer kompletten XML-Struktur für die SPL aus Abbildung 2.2

Diese XML-Struktur verwendet der Nutzer nun, um seine Anforderungen vom SMT-Solver lösen zu lassen. Dazu übergibt er diese dem Prototypen, welcher ihn in den Syntax von JavaSMT umwandelt.

4.2.2 Umsetzen der Constraints mit JavaSMT

Nach der Umwandlung des Feature-Modells und der Configuration-Constraints erstellt der Prototyp eine Formel basierend auf den Klassen des spldev-Frameworks. Diese Formel überführen wir mittels JavaSMT in die Syntax der SMT-Solver. Dabei ist es abhängig vom verwendeten Solver und den verwendeten Formelbestandteilen, ob die Formel in die Syntax des Solvers übersetzt werden kann. Einige der in JavaSMT unterstützten Solver haben beispielsweise Einschränkungen aufgrund der SMT-Logiken, die sie implementieren. So unterstützt der Solver *Princess* keine reellen Zahlen und *SMTInterpol* erlaubt nur die Multiplikation von Konstanten.

In Tabelle 4.1 ist aufgeschlüsselt, welche Logiken die Solver für welche Funktionen implementieren müssen. Werden keine Feature-Attribute oder Aggregationsfunktionen verwendet, wird nur die Kern-Logik von Barrett et al. [2015] benötigt. Werden Feature-Attribute mit ganzzahligen Werten im Modell verwendet, ist es erfordlich, dass der verwendete Solver die Logik `QF_IDL` unterstützt. Für reelle Zahlen ist die Logik `QF_RDL` vonnöten.

Feature-Attribute	Configuration-Constraints	SMTLib Logiken
Keine	Keine	Core
Ganze Zahlen	Keine	Core, `QF_IDL`
Reelle Zahlen	Keine	Core, `QF_RDL`
Ganze Zahlen	Ganze Zahlen	Core, `QF_IDL`
Reelle Zahlen	Reelle Zahlen	Core, `QF_RDL`
Ganze/Reelle Zahlen	Ganze Zahlen	Core, `QF_IDL`, `QF_RDL`
Ganze/Reelle Zahlen	Reelle Zahlen	Core, `QF_IDL`, `QF_RDL`
Ganze/Reelle Zahlen	Ganze/Reelle Zahlen	Core, `QF_IDL`, `QF_RDL`
Ganze/Reelle Zahlen	Ganze/Reelle Zahlen	Core, `QF_IDL`, `QF_RDL`

Tabelle 4.1: Benötigte Logiken der SMTLib [Barrett et al., 2015]

5. Evaluierung

In diesem Kapitel beschreiben wir unsere Forschungsfragen ein und beschreiben einen Versuchsaufbau, der als Grundlage für die Beantwortung der Forschungsfragen dienen soll. Bei der Beschreibung und Diskussion der Ergebnisse gehen wir dabei auf jede Forschungsfrage einzeln ein.

5.1 Forschungsfragen

In diesem Abschnitt definieren wir die Forschungsfragen (**FF**) ein, die wir durch unsere Evaluierung beantworten wollen, um die Tauglichkeit unseres Konzeptes bewerten zu können.

FF 1. *Beeinflusst das Kodieren der Feature-Attribute mittels Werteinschränkung die Größe der Formel für das erweiterte Feature-Modell?*

Im Konzept aus Kapitel 3 definieren wir folgende zusätzliche Formeln, um Feature-Attribute in den Configuration-Constraints verwenden zu können:

- Werden Feature-Modelle anhand unseres Konzeptes eingelesen und in eine aussagenlogische Formel umgewandelt, werden für die enthaltenen Features und die Feature-Attribute pro Feature Variablen erstellt. Aufgrund der im Feature-Modell enthaltenen Definitionen bezüglich der Feature-Attribut-Werte werden für die dazugehörigen Variablen in der Formel zusätzliche Klauseln definiert, die die Werte für diese Variablen einschränken.
- In unserem Konzept führen wir eine Variable für jedes Feature ein, die entsprechend der Auswahl des Features die Werte 0 oder 1 annimmt. Hierzu werden der Formel für das eingelesene Feature-Modell zusätzliche Formeln hinzugefügt. Dieser Variablen nennen wir *Count*.

Durch die zusätzlichen Werteinschränkungen ist die Größe der aussagenlogischen Formel für das Feature-Modell nicht mehr nur von der Anzahl der Features, sondern auch von der Anzahl an unterstützten Aggregationsfunktionen und von den im Modell

enthaltenen Feature-Attributen abhängig. In unserem ersten Experiment untersuchen wir daher für unterschiedliche Feature-Modelle, ob sich das Kodieren anhand des Konzepts aus Kapitel 3 auf die Größe der aussagenlogischen Formel auswirkt.

FF 2. *Wirkt sich die Größe der Formel für ein Modell auf die Zeit aus, die der Solver für Analysen benötigt?* Um Konfigurationen zu einem Feature-Modell zu finden und zu testen, verwenden wir SMT-Solver, denen wir das Feature-Modell als aussagenlogische Formel übergeben. Da unterschiedliche Feature-Modelle unterschiedliche viele Features haben, überprüfen wir mit unserem zweiten Experiment, ob sich erweiterte Feature-Modelle mit unterschiedlicher Feature-Anzahl auf die Zeit auswirken, die der Solver braucht, um eine Lösung für eine Formel eines Feature-Modells zu finden.

Damit der Solver die Formel für das Feature-Modell entgegennehmen kann, wandeln wir diese aus der Syntax des spldev[4]-Frameworks mithilfe von JavaSMT in die Syntax des Solvers um. Da wir erwarten, dass in einer interaktiven Umgebung dieser Prozess häufig in schneller Abfolge durchgeführt wird, erfassen wir zusätzlich zur Zeit zum Finden einer Lösung auch die Zeit, wie lange die Umwandlung der Formel in eine für den Solver geeignete Syntax benötigt. Dies erlaubt uns, eventuelle Bottlenecks zu identifizieren und zu überprüfen, ob die Umwandlung der Formel mittels JavaSMT sich auf den Prozess des Findens einer Lösung auswirkt.

FF 3. *Unterscheidet sich die Performanz des Solvers bei unterschiedlichen Aggregationsfunktionen?* Mit unserem Konzept unterstützen wir verschiedene Aggregationsfunktionen, die der Nutzer in den Configuration-Constraints anwenden kann. Daher untersuchen wir in unserem dritten Experiment, ob sich die Aggregationsfunktionen auf die Zeit auswirken, die der Solver benötigt, um eine Lösung für die Formel eines erweiterten Feature-Modell mit Configuration-Constraints zu finden. Dabei untersuchen wir die in Kapitel 4 definierten Aggregationsfunktionen *avg*, *mul* und *sum*. Für jede Funktion definieren wir für ein Modell ein Configuration-Constraint, welches die Aggregationsfunktion mit einem konstanten Wert vergleicht.
Da auch hier die Formel zunächst aus der Syntax des spldev-Frameworks in eine für den Solver geeignete Syntax überführt wird, protokollieren wir auch die Zeit, die für die Umwandlung mittels JavaSMT benötigt wird.

FF 4. *Unterscheidet sich die Performanz der Solver bei Configuration-Constraints mit einer Aggregationsfunktion und bei Configuration-Constraints mit mehreren Aggregationsfunktionen?* Bei unserem dritten Experiment untersuchen wir, ob sich ein Configuration-Constraint mit einer Aggregationsfunktion auf die Zeit des Findens einer Lösung auswirkt. Da Aggregationsfunktionen jedoch auch miteinander kombiniert werden können, ersetzen wir in unserem vierten Experiment die Konstante in dem Configuration-Constraint aus unserem dritten Experiment durch eine zweite Aggregationsfunktion. Wir untersuchen, ob eine Kombination aus den Aggregationsfunktionen *avg*, *mul* und *sum* sich auf die Zeit, die der Solver benötigt, auswirkt. Dabei kombinieren wir jeweils zwei Aggregationsfunktionen, sodass wir die folgenden Kombinationen untersuchen:

- Arithmetisches Mittel und Arithmetisches Mittel
- Arithmetisches Mittel und Produkt

[4] https://github.com/skrieter/spldev

- Arithmetisches Mittel und Summe
- Produkt und Produkt
- Produkt und Summe
- Summe und Summe

Da auch die Configuration-Constraints bei der Umwandlung mittels JavaSMT verarbeitet werden, protokollieren wir auch hier die Zeit, die die Umwandlung benötigt.

5.2 Verwendete Feature-Modelle und Versuchsaufbau

Um die Forschungfragen aus Abschnitt 5.1 zu beantworten, verwenden wir verschiedene Feature-Modelle. Diese variieren sowohl in der Anzahl der enthaltenen Feature-Attribute als auch in der Anzahl der enthaltenen Feature-Attribute. In diesem Abschnitt gehen wir daher kurz auf die verwendeten Modelle ein. Zusätzlich erläutern wir unser Vorgehen bei den Experimenten.

5.2.1 Modelle

Obwohl es viele frei verfügbare Feature-Modelle ohne Feature-Attribute im Format von FeatureIDE gibt, konnten wir nur wenige Feature-Modelle mit Feature-Attributen ermitteln, die bereits in dem von uns benötigten Format vorlagen. Daher verwenden wir alle gefundenen Feature-Attribute, da diese sich in der Anzahl ihrer Features jeweils um Zehnerpotenzen unterscheiden und damit als Repräsentanten verschiedener Größenordnungen dienen.

Um Feature-Modelle unterschiedlicher Größenordnungen bei der Durchführung der Experimente zu untersuchen, verwenden wir das in dieser Arbeit vorgestellte Modell *webserver* und ein Sandwich-Modell *sandwich*[5] als kleine Modelle, ein Modelle *pc_config*[6] als mitteres und das Feature-Modell zu einem Linux-Kernel *linux*[7] als großes Modell. Diese Modelle liegen bereits im von uns benötigten XML-Format vor. Die Feature-Modelle werden in Tabelle 5.1 nochmals mit der Anzahl der enthaltenen Features und Feature-Attribute aufgelistet.

Das Feature-Modell *webserver* und das Feature-Modell *sandwich* stellen zwei numerische Feature-Attribute bereit und haben 17 beziehungsweise 19 Features. Beim Sandwich-Modell wird das enthaltene Feature-Attribut *Organic Food* nicht berücksichtigt, da diesem der Wertebereich Boolean zugewiesen ist und wir nur Ganzzahlen und reele Zahlen unterstützen (vgl. Kapitel 4).

Das Feature-Modell *pc_config* beschreibt die Möglichkeiten beim Zustammenstellen eines PCs, besitzt 377 Features und ein Feature-Attribut namens *Price*.

[5] Modell unter https://github.com/FeatureIDE/FeatureIDE/blob/master/plugins/de.ovgu.featureide.examples/featureide_examples/ExtendedFeatureModeling/Sandwich/model.xml

[6] Modell unter https://github.com/FeatureIDE/FeatureIDE/blob/master/plugins/de.ovgu.featureide.examples/featureide_examples/ExtendedFeatureModeling/PcConfigurator/model.xml

[7] Modell unter https://github.com/AlexanderKnueppel/is-there-a-mismatch/blob/master/Data/LargeFeatureModels/KConfig/linux-2.6.33.3.xml

Modell	#Features	#Attribute
webserver	17	2
sandwich	19	2
pc_config	377	1
linux	6467	0

Tabelle 5.1: Verwendete Modelle mit Feature- und Feature-Attribut-Anzahl

Das Feature-Modelle *linux* enthält 6467 Features und kein Feature-Attribut. Da durch das Konzept nicht nur erweiterte Feature-Modelle, sondern auch Feature-Modelle ohne Feature-Attribute unterstützt werden sollen, ist dieses Feature-Modell somit auch ein Vertreter für Feature-Modelle ohne Feature-Attribute.

5.2.2 Durchführung

In diesem Abschnitt beschreiben wir unser Vorgehen bei der Durchführung der Experimente, um die Forschungsfragen zu beantworten.

Um die ausgewählten Feature-Modelle zu analysieren, lesen wir sie aus dem XML-Format aus Unterabschnitt 4.2.1 ein und wandeln sie, wie in Kapitel 3 beschrieben, in eine aussagenlogische Formel um. Dabei unterscheiden wir abhängig von der Forschungsfrage zwischen drei verschiedenen Vorgehensweisen.

- Soll ein Feature-Modell oder ein erweitertes Feature-Modell ohne Attribute eingelesen werden, ignorieren wir die im Feature-Modell enthaltenen Feature-Attribute. Dadurch fügen wir auch nicht die dazugehörigen Formeln für die Werteinschränkungen in die Formel für das Feature-Modell ein. Die daraus resultierende aussagenlogische Formel für das Feature-Modell enthält alle Cross-Tree-Constraints, alle Features und deren Beziehungen untereinander.
- Weiterhin lesen wir die Feature-Modelle als erweiterte Feature-Modelle ein. Dabei differenzieren wir nicht zwischen Feature-Modellen mit oder ohne Feature-Attributen. Wird ein Feature-Modell ohne Feature-Attribute eingelesen, resultiert dies in einer Formel, die keine Formeln für Werteinschränkungen enthält.
- Zudem lesen wir Feature-Modelle mit den in ihnen definierten Feature-Attributen ein, generieren jedoch zusätzlich Variablen für die Feature-Auswahl (Count) und fügen der Formel für das Feature-Modell Formeln für die Werteinschränkung dieser Variablen hinzu.
- Bei einigen Forschungsfragen untersuchen wir die Auswirkungen von Aggregationsfunktionen. Da diese in den Configuration-Constraints definiert werden, lesen wir bei der vierten Vorgehensweise nicht nur das erweiterte Feature-Modell mit Attributen und der Zählvariable ein. Zusätzlich dazu fügen wir der Formel für das Feature-Modell weitere Formeln hinzu, die durch die Configuration-Constraints definiert werden.

Für jedes Modell der in Tabelle 5.1 festgehaltenen Modelle erhalten wir damit jeweils drei unterschiedliche Formeln. Um erfüllende Belegungen für diese Formeln zu finden,

Modell	#Feat	#Lit	#Attr	#Var
webserver (Feature-Modell)	17	50	0	17
webserver (Erw. Feature-Modell)	17	271	2	82
webserver (Erw. Feature-Modell mit Count)	17	340	2	100
sandwich (Feature-Modell)	19	58	0	19
sandwich (Erw. Feature-Modell)	19	517	2	154
sandwich (Erw. Feature-Modell mit Count)	19	594	2	174
pc_config (Feature-Modell)	377	3277	0	377
pc_config (Erw. Feature-Modell)	377	8071	1	1787
pc_config (Erw. Feature-Modell mit Count)	377	9580	1	2165
linux (Feature-Modell)	6467	280149	0	6467
linux (Erw. Feature-Modell)	6467	280149	0	6467
linux (Erw. Feature-Modell mit Count)	6467	306018	0	12935

Tabelle 5.2: Verwendete Modelle nach Feature-, Literal-, Feature-Attribut und Variablen-Anzahl nach der Umwandlung

verwenden wir einen SMT-Solver. Dazu wandeln wir die Formeln mittels JavaSMT und dessen Schnittstelle zum Z3-Solver [Moura und Bjørner, 2008] in die Syntax des Z3-Solvers um. Dabei erfassen wir die Zeit, die für die Umwandlung in den Syntax des Solvers benötigt wird. Da bei der Anwendung unseres Konzeptes alle benötigten Variablen direkt beim Erstellen des Modells in die Formel integriert werden, ist der Schritt der Umwandlung relevant. Bei jeder Analyse auf dem Feature-Modell muss das Feature-Modell von der internen Struktur zuerst in die Syntax des Solvers, in unserem Fall Z3, umgewandelt werden, damit der Solver für die Analyse verwendet werden kann. Daher ist die Zeit, die dieser Prozess in Anspruch nimmt für unsere Forschungsfragen relevant.

Weiterhin lassen sich viele der in Kapitel 2 genannten Analysen auf die Erfüllbarkeitsanalyse einer Variablenbelegung reduzieren. Daher messen wir die Zeit, die der Solver für das Finden einer möglichen Lösung für die ihm übergebene Formel benötigt.

5.2.3 Technische Voraussetzungen

Die Experimente wurden auf einem Windows 10-System mit einer 64-Bit-Architektur durchgeführt. Der verwendete Rechner läuft mit einem Intel Core i5 Prozessor mit 1 Socket und 4 Kernen. Der Maschine stehen 8GB RAM zur Verfügung, wobei der Java Virtual Maschine bei der Ausführung ein Maximallimit von 2GB angegeben wurden. Die Tests für die einzelnen Experimente wiederholen wir dabei 100 Mal.

5.3 Ergebnisse

In diesem Abschnitt gehen wir auf die Ergebnisse unserer Experimente ein und diskutieren diese anhand unserer Forschungsfragen.

FF1. In unserem ersten Experiment untersuchen wird die Feature-Modelle in Bezug auf die Anzahl der Literale in der Formel für das Feature-Modell. Dabei betrachten wir für jedes der in Tabelle 5.1 aufgeführten Modelle, wie sich die Anzahl der Literale in

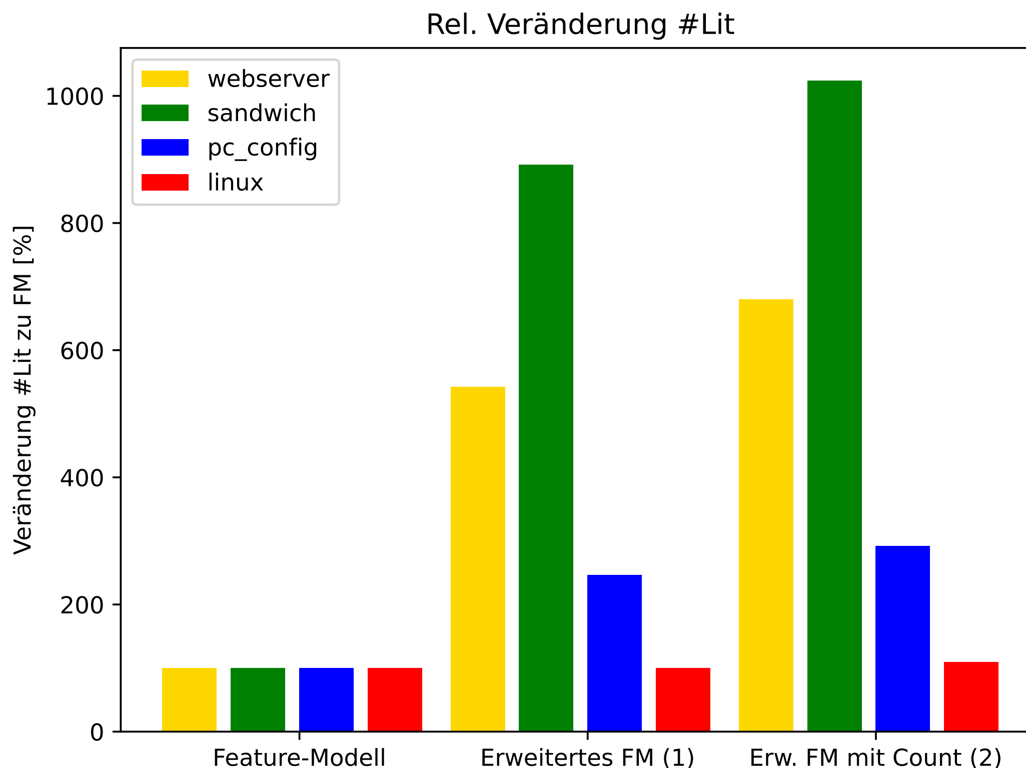

Abbildung 5.1: Anzahl der Literale verschiedener Feature-Modelle in Bezug auf das jeweilige Modell ohne Attribute und Count (FF1)

dem Modell ändert, wenn *(1)* die Attribute aus dem Feature-Modell in die Formel mit übernommen werden und wenn *(2)* zusätzlich zu den Attributen auch Zählvariable *count* mit in die Formel übernommen wird.

Die Ergebnisse des Experiments sind in Tabelle 5.2 aufgeführt. Für jedes Modell erfassen wir die Anzahl der Features im Modell, die Anzahl an Feature-Attributen, die bei dem jeweiligen Modell eingelesen wurden, die Variablen in der Formel (inklusive Features, -Attribute und Count-Variablen) und letztendlich die Anzahl an Literalen in der Formel. Die relative Anzahl an Literalen in Bezug auf das jeweilige Feature-Modell ohne Attribute und Count-Variablen ist dabei in Abbildung 5.1 dargestellt.

Relativ zur Durchführung mit Feature-Modell ohne Attribute ist bei *(1)* und *(2)* zu erkennen, dass die Anzahl an Literalen jeweils größer ist. Dies lässt sich auf den Anstieg an den zu kodierenden Variablen zurückführen. Bei den erweiterten Feature-Modellen *(1)* fügen wir zu der Formel für das Modell zusätzliche Variablen für Feature-Attribute ein, deren Werte wir mittels zusätzlicher Formeln einschränken. Da im Modell *linux* keine Feature-Attribute definiert sind, ist hier kein Anstieg in den Literalen zu verzeichnen. Bei den kleineren Feature-Modellen steigt die Literalanzahl jedoch auf das fast 5,5-fache (*webserver*) beziehungsweise das fast 9-fache (*pc_config*) von der Anzahl Literalen beim Feature-Modell. Beim Feature-Modell mittlerer Größe steigt die Literalanzahl bei (1) relativ gesehen weniger als bei den kleineren Modellen, jedoch verdoppelt sich die Anzahl der Literale auch hier. Dies führen wir darauf

zurück, dass bereits in der Formel für das Feature-Modell ohne Attribute und Count mehr Literale enthalten sind.

Der Anstieg der Literale im Vergleich zum Feature-Modell ohne Attribute ist dabei linear abhängig zu der Anzahl an Features mit Feature-Attributen und der Anzahl an Aggregationsfunktionen. Der Anstieg wird dabei durch folgende Faktoren herbeigeführt:

- Für jedes Feature mit Feature-Attribut werden Formeln erzeugt, die den Wert der dazugehörigen Variable einschränken.
- Diese Einschränkungen werden für alle unterstützen Aggregationsfunktionen durchgeführt.
- Die Einschränkungen werden für den Fall der Selektion und der Deselektion des Features getroffen.

Bei kleineren Feature-Modellen ist dabei die Auswirkung auf die Größe der Formel größer als bei größeren Modellen.

Wird die Count-Variable inklusive aller Hilfsvariablen zur Formel für ein (erweitertes) Feature-Modell hinzugefügt, steigt auch hier die Anzahl der Literale im Vergleich zur Anzahl an Literalen bei der Formel für das Feature-Modell ohne Count-Variable. In Abbildung Abbildung 5.1 wird die Count-Variable zum erweiterten Feature-Modell hinzugefügt. Der Anstieg ist dabei deutlich geringer als bei *(1)*, da die Anzahl der hinzugefügten Literale nur abhängig ist von der Feature-Anzahl. Hierbei ist der Anstieg der Literale bei kleineren Modellen relativ gesehen größer als bei größeren Modellen.

Insgesamt steigt die Anzahl der Literale in den Formel durch die Kodierung aus Kapitel 3 an. Vor allem in Umgebungen mit begrenzten Ressourcen kann dies bei der Ausführung mittels Solvern dazu führen, dass die Solver eine längere Zeit für die Arbeit mit den Formel benötigen. Dies trifft sowohl auf kleinere, als auch größere Modelle zu, da bei größeren Modellen die absoluten Änderungen in der Literalanzahl allein durch die Anzahl an Features herbeigeführt werden. Hier können Optimierungen in der Formel Abhilfe schaffen, indem Literale entfernt beziehungsweise umstrukturiert werden.

FF2. In unserem zweiten Experiment untersuchen wir *(1)* die Zeit, die der Prototyp für die Umwandlung der aussagenlogischen Formel in den Syntax des Solvers und *(2)* die Zeit, die der Solver zum Finden einer Lösung für diese Formel benötigt. Die Suche aller Lösungen betrachten wir nicht, da das Finden aller Lösungen nicht trivial ist und die Durchführung mit einem Timeout nach 30 Sekunden bereits bei kleinen Modellen wie *webserver* nicht erfolgreich war.
Im Experiment betrachten wir nur die Modelle *webserver*, *sandwich* und *pc_config*. Bei der Durchführung kam es bei dem Modell *linux* zu einer StackoverflowException bei der Umwandlung mittels JavaSMT. Die Implementierung der Umwandlung in spldev iteriert rekursiv durch die baumartige Struktur und die Anzahl der zu bearbeitenden Knoten bei den Attributen wurde zu hoch.

Die Zeiten für *(1)* und *(2)* sind in Abbildung 5.2 in Relation zur Zeit für die Ausführung für das Feature-Modell ohne Attribute und Count dargestellt. In der

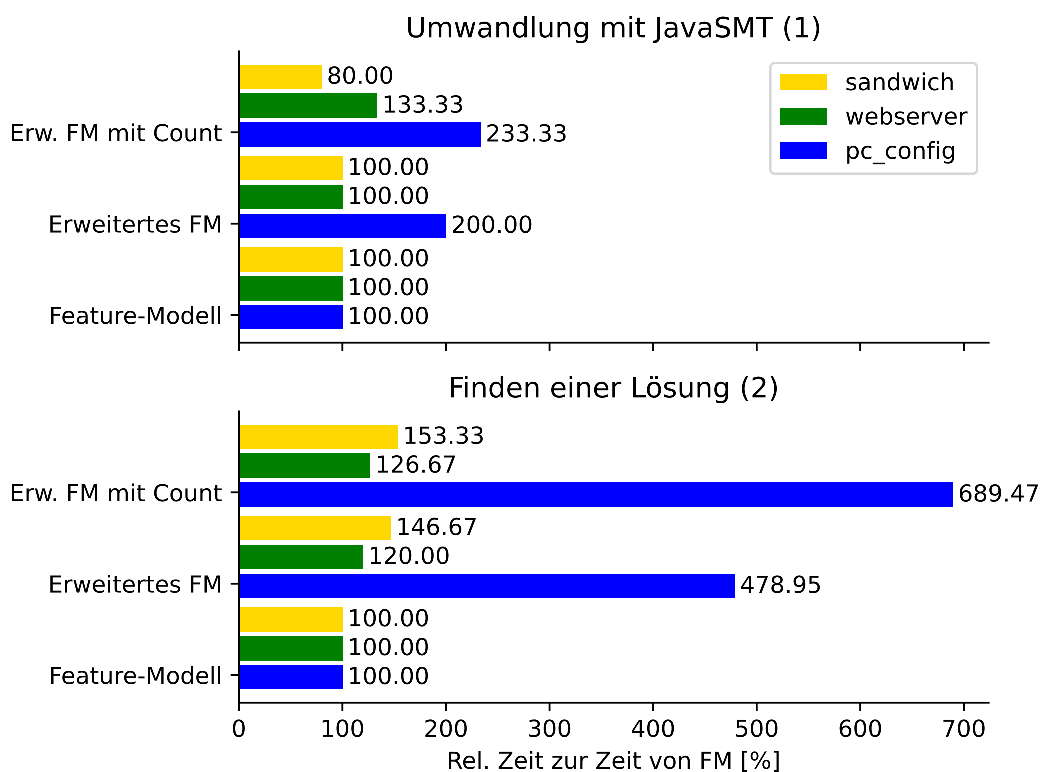

Abbildung 5.2: Zeiten für die Umwandlung von Feature-Modellen und das Finden einer Lösung mit Z3 bezogen auf das jeweilige Feature-Modell ohne Attribute und Count

Abbildung sind die Median-Werte dargestellt, damit Ausreißer in den Daten die Ergebnisse nicht verzerren. Bezüglich *(1)* ist erkennbar, das sich die Zeit beim mittleren Modell ähnlich verhält wie die dazugehörige Änderung der Literale bei **FF1**: Je mehr Literale in der Formel enthalten sind, desto mehr Zeit wird für die Umwandlung der Formel mittels JavaSMT benötigt. Bei den kleineren Modellen *webserver* und *sandwich* ist dies durch die geringe Feature-Anzahl nicht zu erkennen. Beim Modell *sandwich* ist zudem beim erweiterten Feature-Modell mit der Count-Variable erkennbar, dass die Umwandlung nur 80 Prozent der Zeit der Umwandlung des Feature-Modells ohne Attribute und Count-Variable benötigt hat. Dies führen wir jedoch auf eine unterschiedliche Auslastung des Testsystems zu Zeiten der Durchführung zurück, da die absoluten Werte bei *(1)* beim Modell *sandwich* zwischen 3 und 5 Millisekunden schwanken. Für das Modell *pc_config* verdoppelte sich die benötigte Zeit für die Umwandlung, sobald Attribute berücksichtigt werden. Dies führen wir auf die Anzahl an Features zurück, für deren Variablen in der Formel Werteinschränkungen vorgenommen werden.

Beim Finden einer Lösung *(2)* verlängert sich die Ausführung, sobald in der Formel für das jeweilige Feature-Modell die Count-Variable oder Attribute berücksichtigt werden. Mit der Anzahl der Literale steigt also auch die Zeit der Durchführung. Für jedes untersuchte Modell ist die Laufzeit bei dem Feature-Modell am geringsten, beim erweiterten Feature-Modell mit der Count-Variable am höchsten und die Laufzeit für das erweiterte Feature-Modell ohne Count-Variable liegt dazwischen. Dies ist unabhängig von der Feature-Anzahl.

Während beim Modell *sandwich* die Feature-Attribute für jedes Feature definiert sind, sind bei *webserver* nur einige Feature mit den Feature-Attributen versehen. Daher ist die relative Zeit bei *sandwich* entsprechend höher als bei *Webserver*, obwohl die Feature-Anzahl nahezu identisch ist.

Einen merklichen Unterschied zu den Werten für die kleinen Modellen gibt es bei den Werten zu Modell *pc_config*. Hier ist die Erhöhung der Laufzeit bei dem erweiterten Feature-Modell auf fast das 5-fache angestiegen. Nimmt man die Count-Variable dazu, steigt die Laufzeit sogar um fast das 6-fache an. Betrachten wir die absoluten Werte für *pc_config*, erhalten wir die folgenden Mediane:

- Zeit beim Feature-Modell: 19.0 ms
- Zeit beim erweiterten Feature-Modell: 91.0 ms
- Zeit beim erweiterten Feature-Modell mit Count: 131.0 ms

Diese Werte lassen darauf schließen, dass die Laufzeit für das Finden einer Lösung mit der Größe der zu lösenden Formel anwächst. Für ein sehr großes Modell kann es daher passieren, dass das Finden einer Lösung nicht mehr praktikabel ist. Für kleine Modelle ist dies nicht der Fall.

FF3. Unser drittes Experiment besteht darin, dass wir für verschiedene Aggregationsfunktionen betrachten, ob sie sich auf die Performanz bei der Umwandlung und auf die Performanz des Solvers beim Finden einer Lösung auswirken. Auch hier führen wir die Tests mit 100 Iterationen durch, um etwaige Schwankungen in der Systemauslastung zu kompensieren. Zudem verwenden als Ergebnis den Median der gemessenen

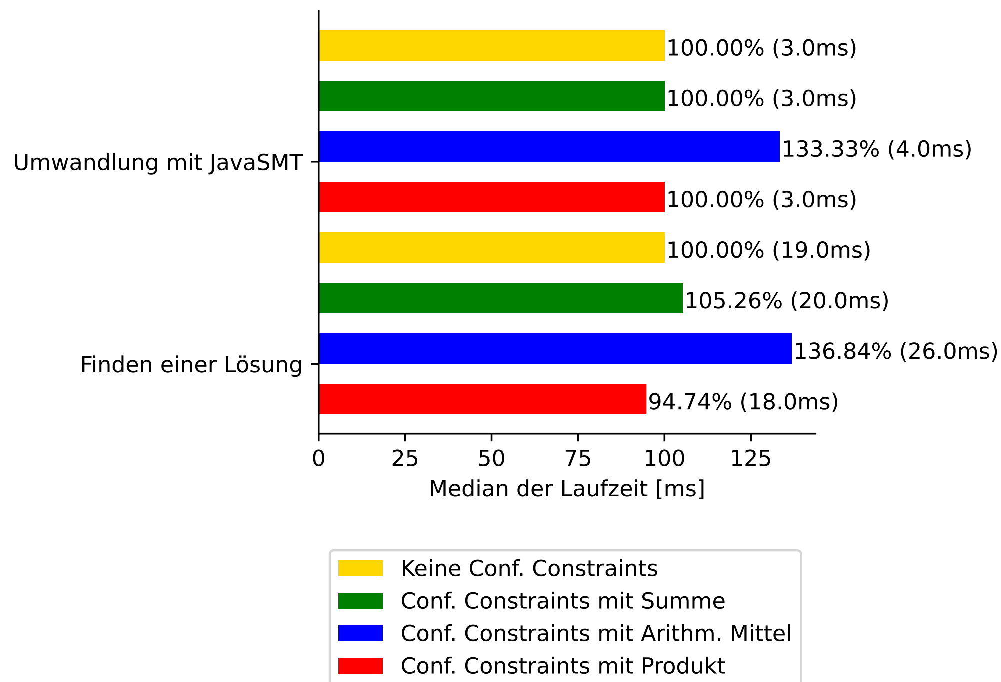

Abbildung 5.3: Zeiten für das Umwandeln und Finden von Lösungen für das Modell *webserver* mit Configuration-Constraints mit versch. Aggr.-Funktionen relativ zur Zeit ohne Configuration-Constraints

Werte, um Ausreißer herauszufiltern. Da wir für unsere Aggregationsfunktionen sowohl die Attribute als auch den Count benötigen, verwenden wir das erweiterte Feature-Modell mit Count-Variable als Grundlage für unsere Untersuchungen. Die Configuration Constraints besteht dabei aus einer einzigen Einschränkung, die eine der Aggregationsfunktionen mit einem konstanten Wert vergleicht. Sie besitzt dabei folgende Struktur:

$$aggr(attr) < k$$

Durchgeführt haben wir die Tests anhand der Modelle *webserver*, *sandwich* und *pc_config*. Die Ergebnisse für *webserver* sind in Abbildung 5.3, die Ergebnisse für *sandwich* in Abbildung 5.4 und die Ergebnisse für *pc_config* in Abbildung 5.5 dargestellt. Die Messergebnisse sind jeweils in Relation zur Laufzeit gestellt, die für das Bearbeiten des Modells ohne Configuration-Constraints benötigt wurde.

Während wir bei den kleinen Modellen für alle implementierten Aggregationsfunktionen Ergebnisse erzielt haben, führte die Anwendung des arithmetischen Mittels beim Modell *pc_config* zu einem Timeout. Daher sind für diese Aggregationsfunktion in den Ergebnissen für *pc_config* keine Ergebnisse enthalten.

Es ist erkennbar, dass sich bei den kleinen Modellen die Aggregationsfunktionen nicht stark auf den Umwandlungsprozess mithilfe von JavaSMT auswirken. Zwar zeigen die Messungen bei *webserver* bei der Umwandlung des Modells mit einer

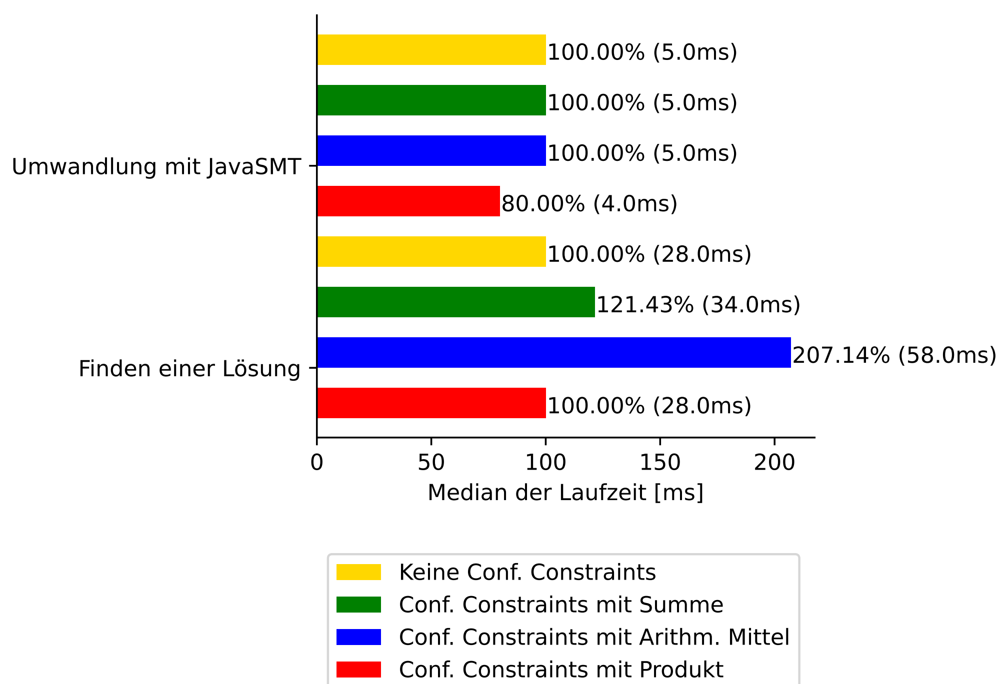

Abbildung 5.4: Zeiten für das Umwandeln und Finden von Lösungen für das Modell *sandwich* mit Configuration-Constraints mit versch. Aggr.-Funktionen relativ zur Zeit ohne Configuration-Constraints

Configuration-Constraints mit dem arithmethischen Mittel eine Abweichung von 33,3 Prozent, diese ist aber in Hinblick auf die absoluten Werte (1ms Unterschied) vernachlässigbar. Genauso verhält es sich beim Modell *sandwich*. Hier führt bei der Umwandlung des Modells ein Unterschied von 1ms Laufzeit dazu, dass die relative Abweichung 20 Prozent beträgt.

Beim Finden einer Lösung für die Formel zu *webserver* ist bei der Laufzeit für die Aggregationsfunktionen Summe und Produkt kein großer Unterschied zur Laufzeit für die Formel ohne Constraints zu erkennen. Auch hier beträgt die Abweichung der Laufzeiten $\pm$ 1ms und ist damit vernachlässigbar. Eine größere Abweichung ist beim arithmetischen Mittel als Aggregationsfunktion in den Configuration Constraints zu erkennen. Hier dauert die Lösungssuche im Vergleich zu den anderen Aggregationsfunktionen ca. 37 Prozent länger. Durch die Kodierung des arithmetischen Mittels als die Differenz zweier Summen (Summe aller Werte für das Feature-Attribut und Count) werden hier mehr Operationen durchgeführt als bei den anderen Aggregationsfunktionen, wodurch sich die Laufzeit verlängert. Bestätigt wird dies auch durch das Modell *sandwich*. Hier ist die Laufzeit sogar um mehr als 100 Prozent länger, wenn in den Configuration-Constraints das arithmetische Mittel als Aggregationsfunktion verwendet wird.

Ansonsten weisen die Laufzeiten für das Modell *sandwich* ähnliche Tendenzen wie die Laufzeiten des Modells *webserver* auf. So sind die auftretenden Unterschiede in der Laufzeit bei der Umwandlung vernachlässigbar gering. Beim Finden einer Lösung für die Modelle mit der Aggregationsfunktion Summe ist die Laufzeit im

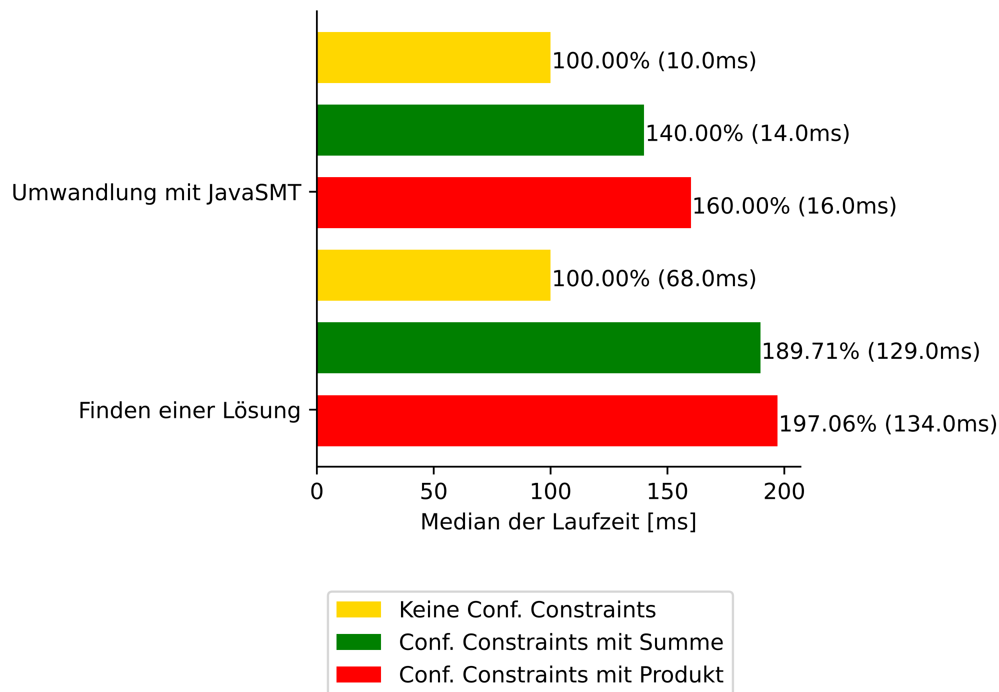

Abbildung 5.5: Zeiten für das Umwandeln und Finden von Lösungen für das Modell *pc_config* mit Configuration-Constraints mit versch. Aggr.-Funktionen relativ zur Zeit ohne Configuration-Constraints

Vergleich zum gleichen Prozess ohne Aggregationsfunktion leicht erhöht und mit einem Unterschied von 21 Prozent höher als beim Modell *webserver*. Dies können wir darauf zurückführen, das beim Modell *sandwich* jedes Feature ohne untergeordnete Features das verwendete Feature-Attribut *Price* aufweist, während die *Latenz* im Modell *webserver* bei weniger Features vorhanden ist.

Beim Modell *pc_config* ist bei den Laufzeiten für die Umwandlung der Modelle mittels JavaSMT im Gegensatz zu den kleinen Modellen ein Anstieg zu erkennen, sobald eine der Aggregationsfunktionen für die Summe oder das Produkt verwendet wird. Dies führen wir auf die Feature-Anzahl im Modell zurück. Durch die im Vergleich zu den kleinen Modellen höhere Anzahl an Features werden die gleichen Operationen für die Umwandlung für mehr Features ausgeführt, wodurch sich die Laufzeit erhöht. Aber obwohl im Vergleich zu *webserver* und *sandwich* die Feature-Anzahl bei *pc_config* um mehr als das 20-fache höher ist, ändert sich die Laufzeit im Verhältnis zu den kleinen Modellen nur auf das 5-fache. Auch wenn wir die Änderung zur Laufzeit für das Modell *pc_config* ohne Aggregationfunktionen betrachten, ist der Anstieg gering. Hier verzeichnen wir einen 60-prozentigen Anstieg der Laufzeit für Produkt und Summe. Beim Finden einer Lösung für *pc_config* steigt die Laufzeit bei der Verwendung von Aggregationsfunktionen im Verhältnis zur Laufzeit ohne Configuration-Constraints um ca. 90 bis 100 Prozent an.

Wir erkennen, dass sich einzelne Aggregationsfunktionen in den Configuration-Constraints bei kleinen Modellen wie *webserver* kaum auf die Laufzeiten der Umwand-

lung mittels JavaSMT auswirken. Bei Modellen wie *pc_config* verzeichnen wir dagegen einen Anstieg der Laufzeit, sobald Aggregationsfunktionen verwendet werden.

Auch beim Finden einer Lösung sind die Auswirkungen von einzelnen Aggregationsfunktionen in Configuration-Constraints auf die Laufzeit bei kleinen Modellen gering. Eine Ausnahme stellt dabei das arithmetische Mittel dar, welches die Laufzeit auch verdoppeln kann (vgl. Abbildung 5.4). Anzahl der vorhanden Feature-Attribute im Feature-Modell sind dabei ein Faktor, der die Laufzeit erhöht. Bei Modellen wie *pc_config* führte die Verwendung der Aggregationsfunktion für das arithmetische Mittel zu einem Timeout. Configuration-Constraints mit Produkt oder Summe erhöhen die Laufzeit beim Finden einer Lösung.

FF4. In unserem vierten Experiment wiederholen wir das dritte Experiment, ändern jedoch die Configuration-Constraints. Um zu überprüfen, ob sich Configuration-Constraints bestehend aus mehreren Aggregationsfunktionen auf die Performanz des Servers auswirken, spezifizieren wir jeweils ein Constraints, die zwei Aggregationsfunktionen miteinander vergleicht. Dadurch muss der Solver mehr der von uns erstellten Variablen auswerten und diesen Werte zuweisen.

Für die Modelle *webserver*, *sandwich* und *pc_config* erstellen wir jeweils Configuration-Constraints die zwei Aggregationsfunktionen verwenden, um Feature-Attribute auszuwerten. Bei *webserver* verwenden wir dabei die beiden verfügbaren Feature-Attribute. Bei *pc_config* beinhaltet das Modell nur ein Feature-Attribut, weshalb wir dieses doppelt verwenden. Bei *sandwich* sind die beiden Datentypen der Feature-Attribute unterschiedlich. Wir verwenden daher nur eines der Feature-Attribute. Für *webserver* erhalten wir die folgenden Kombinationen in den Configuration-Constraints:

- avg(Latenz) $<$ avg(MAnf)
- avg(Latenz) $<$ mul(MAnf)
- avg(Latenz) $<$ sum(MAnf)
- mul(Latenz) $<$ sum(MAnf)
- sum(Latenz) $<$ sum(MAnf)

Bei den Untersuchungen mit dem Modellen *sandwich* und *pc_config* ersetzen wir beide Feature-Attribute durch das Feature-Attribut *Price*.

Die Ergebnisse der Untersuchungen sind in Abbildung 5.6 für das Modell *webserver*, in Abbildung 5.7 für das Modell *sandwich* und in Abbildung 5.8 für das Modell *pc_config* dargestellt. Auch hier werden die Laufzeiten in Relation zu den Laufzeiten für die jeweiligen Modelle ohne Configuration-Constraints gesetzt, um die Auswirkungen der Kombination von Aggregationsfunktionen aufzuzeigen.

Wie wir erwartet haben, zeichnet sich bei der Umwandlung beim Modell *webserver* ab, dass die Kombination zweier arithmetischer Mittels als Aggregationsfunktionen die Laufzeit am meisten verlängern. Unerwartet ist jedoch, dass andere Kombinationen mit dem arithmetischen Mittel gleichermaßen performant laufen wie die Umwandlung des Modells ohne Configuration-Constraints. Beim Modell *sandwich* ist die Umwandlung des Modells mit Configuration-Constraints sogar schneller als

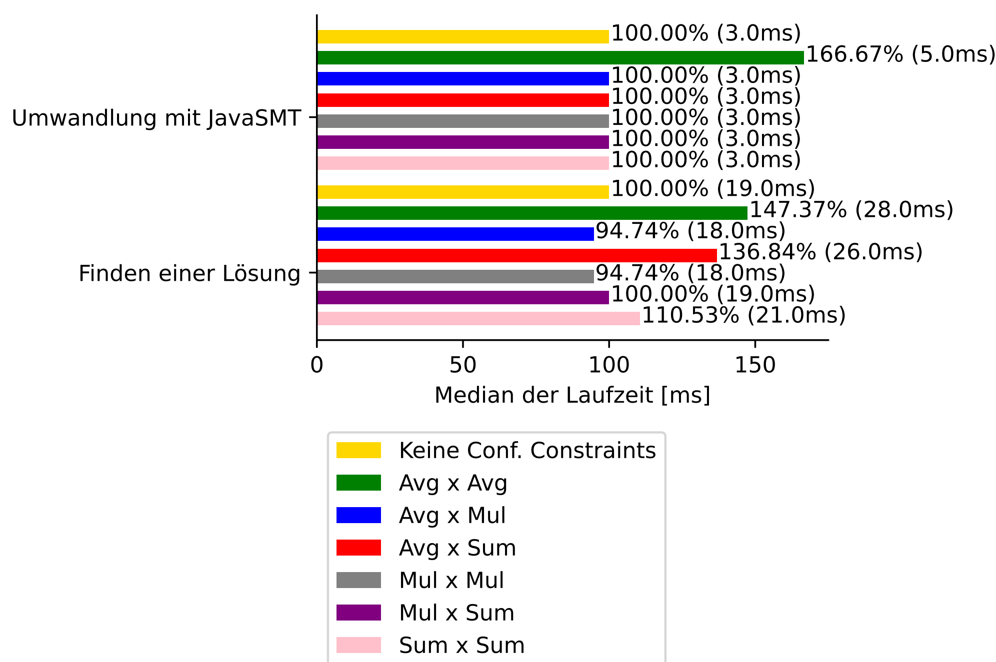

Abbildung 5.6: Zeiten für das Umwandeln und Finden von Lösungen für das Modell *webserver* mit Configuration-Constraint mit Kombinationen von Aggregationsfunktionen relativ zur Zeit ohne Configuration-Constraints

ohne Configuration-Constraints. Durch Wiederholungen der Messungen konnten das Ergebnis mehrfach bestätigen. Erst beim größeren Modell *pc_config* trat der von uns erwartete Fall ein, dass die Laufzeit der Umwandlung für das Modell in Verbindung mit den Configuration-Constraints höher ist, als die Laufzeit für das Modell ohne die Constraints.

Betrachten wir die Laufzeit für das Finden von Lösungen für die einzelnen Modelle mit Configuration-Constraints, treffen keine unserer Erwartungen zu. Auch hier nahmen wir an, dass die Kombination mit einem arithmetischen Mittel die längste Laufzeit aufweisen würde. Eingetreten ist dies nur für das Modell *webserver*, sobei auch hier die Kombination des arithmetischen Mittels mit dem Produkt eine kleiner Laufzeit aufweist als die Laufzeit für das Modell ohne Configuration-Constraints.

Für *sandwich* und *pc_config* lassen sich durch unsere Evaluierung keine klaren Tendenzen für die Auswirkung der Aggregationsfunktionen erkennen. Hier sind die Laufzeiten selbst bei einem größeren Modell teilweise bis zu 50 Prozent geringer als die von uns gewählte Referenzlaufzeit (vlg. Abbildung 5.6). Teilweise weisen die Laufzeiten für Kombinationen von Aggregationsfunktionen bei dem gleichen Modell auch eine Laufzeit von 300 Prozent des Referenzwertes auf. Wir vermuten, dass dies auf Art der Configuration-Constraints zurückzuführen ist. Während bei *webserver* zwei unterschiedliche Feature-Attribute verbunden werden, wird bei *pc_config* das Feature-Attribut *Price* mit sich selbst verbunden. So entstehen Configuration-Constraints wie

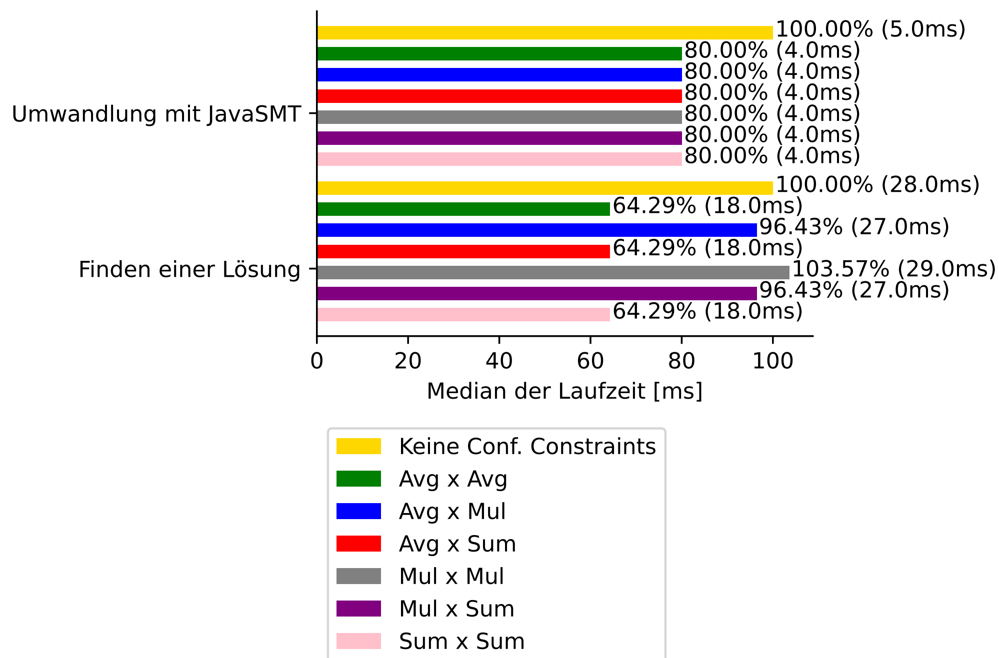

Abbildung 5.7: Zeiten für das Umwandeln und Finden von Lösungen für das Modell *sandwich* mit Configuration-Constraint mit Kombinationen von Aggregationsfunktionen relativ zur Zeit ohne Configuration-Constraints

- avg(Price) $<$ avg(Price)
- avg(Price) $<$ mul(Price)
- avg(Price) $<$ sum(Price)
- mul(Price) $<$ mul(Price)
- mul(Price) $<$ sum(Price)
- sum(Price) $<$ sum(Price)

Da einige der Constraints bereits zu einem Widerspruch führen, kann es für die Formel somit keine Lösung geben. So kann ein beliebiger Wert x nicht kleiner als sich selbst sein. Die Formel $x < x$ ist damit immer unwahr. Wir vermuten weiterhin, dass es im verwendeten Solver bei der Lösung der Menge an übergebenen Formeln eine Überprüfung gibt, durch die der Solver die Bearbeitung der Anfrage frühzeitig abbrechen kann.

Wir können nicht klar feststellen, wieso die unterschiedlichen Kombinationen von Aggregationsfunktionen sich auf die Laufzeit bei der Umwandlung mittels JavaSMT und dem Finden einer Lösung in der festgestellten Art auswirken. Wir stellen jedoch sicher fest, dass die zusätzlichen Formeln abhängig vom Modell und der gewählten Aggregationsfunktionen sowohl zu einer Verringerung als auch einer Verlängerung der Laufzeit führen können.

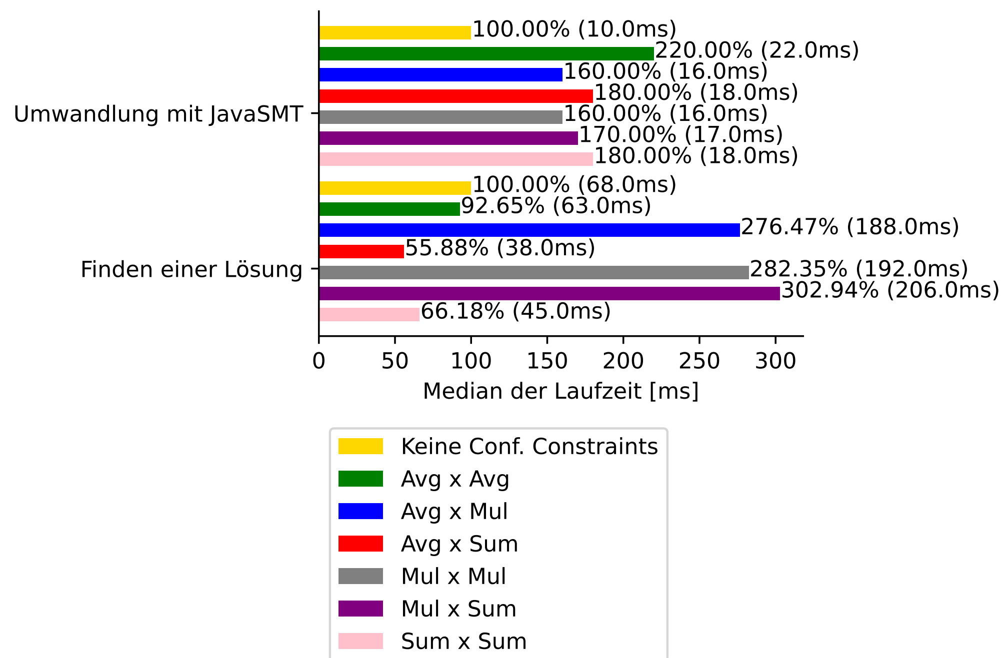

Abbildung 5.8: Zeiten für das Umwandeln und Finden von Lösungen für das Modell *pc_config* mit Configuration-Constraint mit Kombinationen von Aggregationsfunktionen relativ zur Zeit ohne Configuration-Constraints

5.4 Angriffspunkte

Für unsere Evaluierung haben wir verschiedene Angriffspunkte identifiziert, welche wir in diesem Abschnitt beschreiben. Dabei unterteilen wir diese nach Bedrohungen der internen und der externen Validität (vgl. [Wohlin et al., 2012]).

5.4.1 Bedrohungen der internen Validität

Einfachere Berechnung des arithmetischen Mittels. Um das arithmetische Mittel auf einer Menge an Zahlen zu bilden, wird die Summe der Zahlen gebildet und durch die Anzahl der Zahlen geteilt. Das gleiche Prinzip verwenden wir beim arithmetischen Mittel als Aggregationsfunktion. Wir summieren die Werte aller Feature-Attribute auf und teilen durch die Anzahl der ausgewählten Features. Dabei verwenden wir für die Anzahl der ausgewählten Features die Count-Variable. Um die Count-Variable zu in ihrem Wertebereich auf diese Anzahl einzuschränken, führen wir für jedes Feature eine Variable ein, der durch Einschränkungen abhängig von der Auswahl des Features den Wert 1 oder den Wert 0 annehmen kann. Alle diese Werte summieren wir auf, um auf die Anzahl der ausgewählten Features zugreifen zu können, und schränken den Wert der Count-Variable auf diese Summe ein. Die Implementierung erfordert dadurch das Hinzufügen einer mit der Feature-Anzahl linear wachsenden Anzahl an zusätzlichen Variablen für den Solver.

Wenn Features nun das übergebene Attribut nicht besitzen, werden sie durch die Werteinschränkung der Count-Variable im Wert für Count berücksichtigt, die Summe

der Feature-Attributwerte berücksichtigt diese Features jedoch nicht, da der Attributwert als 0 behandelt wird. Dies führt mitunter dazu, dass bei der Verwendung des arithmetischen Mittels vom Nutzer unerwartete Ergebnisse auftreten.

Unterschiedliche Auslastung des Systems. Da bei der Ausführung des Systems auch andere Programme auf die Ressourcen des Systems zugreifen, kann es bei den Messungen zu unerwarteten Abweichungen kommen. Um diesen Umstand zuvorzukommen, haben wir die Messungen in mehreren Iterationen durchgeführt. Jeder Test wurde dabei 100 Mal hintereinander durchgeführt und auf den erfassten Messwerten der Median angewendet.

Externer Solver. Durch die Verwendung eines externen Solvers (Z3), welcher über JavaSMT eingebunden und aufgerufen wird, verwenden wir eine zusätzliche Schnittstelle für das Finden von Lösungen. Dabei greifen wir zwecks der Plattformunabhängigkeit nicht direkt auf den Solver zu, sondern verwenden JavaSMT, welches die Anfragen für den Solver umwandelt. Um dem vorzubeugen, wurden die Experimente mehrmals wiederholt. Die Messwerte lieferten bei mehreren Durchläufen der Experimente die gleichen Ergebnisse und konnten für die kleinen Modelle auch verifiziert werden.

5.4.2 Bedrohungen der externen Validität

Realistische Beispiele. Bei den verwendeten Modellen handelt es sich um Anwendungsfälle, die sich an realistischen Beispielen orientieren. Bei der Auswahl der Modelle haben wir darauf geachtet, zusätzlich zu dem Modell aus dieser Arbeit bereits existierende Modelle zu verwenden.

Abhängigkeit vom Solver. Alle Ergebnisse wurden aufgrund der Gegebenheiten beim Testsystem in Verbindung mit der Verwendung des Z3-Solvers erfasst. Für andere Solver können die Ergebnisse abhängig von der Funktionsweise des Solvers variieren. Dies wird vor allem bei den Ergebnissen in Abbildung 5.8 deutlich, da es hier vom Z3-Solver abhängig, in welcher Reihenfolge die Formelbestandteile abgearbeitet werden.

6. Verwandte Arbeiten

In diesem Kapitel gehen wir auf andere Arbeiten, deren Ausarbeitungen mit unserer Arbeit in Verbindung stehen. Wir orientieren uns dabei an Arbeiten, die *(1)* durch ihr Konzept Feature-Attribute in Constraints erlauben und die *(2)* den Nutzer während des Prozesses des Konfigurierens unterstützen.

Henneberg [2011] stellen in ihrer Arbeit eine Möglichkeit dar, um erweiterte Feature-Modelle mit ganzzahligen Feature-Attributen auf Instanzen von Pseudo-Boolean-SAT-Problemen umzuwandeln. Sie definieren dabei eine eigene Grammatik für das Feature-Modell und die Modeling Constraints. Sie erlauben dabei die Verwendung von Vergleichen auf Gleichheit und Ungleichheit auf Basis der Attribute in den Modeling Constraints, um die Funktionalität der Feature-Modelle zu erweitern. In den Constraints ist es den Anwendern möglich, die Feature und Feature-Attribute zusammen mit konstanten Ganzzahlen zu addieren und subtrahieren. In unserer Arbeit erweitern wir die Funktionsweise der zusätzlichen Constraints um das Produkt über einem Feature-Attribut, reduzieren den Nutzeraufwand durch vordefinierte Funktionen und erlauben es, dass der Nutzer die Constraints während des Konfigurierens für seinen Anwendungsfall definiert.

Auch Karataş et al. [2013] erlauben zusätzlich zu Booleschen Operationen weitere arithmetische Operationen in den Constraints. Durch beispielsweise Summen können Feature-Attribute aufsummiert werden. Die Ergebnisse dieser Operationen verwenden Karataş et al. in Vergleichen auf Ungleichheit beziehungsweise Gleichheit in den Constraints. Sie beziehen sich im Gegensatz zu unserer Arbeit auf Modeling Constraints.

Auf ähnliche Weise gehen Benavides et al. [2005] vor, um Feature-Attribute im Modell zu aggregieren. Nutzer definieren die Feature-Attribute für die Features ohne untergeordnete Features im Modell und weisen diesen Werte zu. Die Werte der Feature-Attribute werden im jeweiligen übergeordneten Features durch Operationen wie beispielsweise Summen aggregiert. Der resultierende Wert ist damit der Wert des Feature-Attributes für das übergeordnete Feature. Sind die Werte bis hin zum Wurzel-Feature aggregiert, finden die im Wurzel-Feature aggregierten Werte

Anwendung in Filtern. Die Filter fügen einzelne Constraints mittels Gleichheit oder Ungleichheit während des Prozesses des Konfigurierens hinzu und erlauben somit die Einschränkung der validen Konfigurationen für ein Feature-Modell. Während Benavides et al. die Aggregation dabei dem Nutzer beim Modellieren überlassen, wählt bei unserem Vorgehen der Nutzer die Aggregation erst bei der Definition der Configuring Constraints aus. Zudem untersuchen Benavides et al. die Anwendung auf Basis von CSP-Solvern, während diese Arbeit sich mit SMT-Solvern beschäftigt.

Die Anwendung von Filtern ist nicht die einzige Möglichkeit, Feature-Attribute während der Konfiguration zu verwenden. So definieren Sprey et al. [2020] eine Analyse, in der sie für eine ausgewählte (Teil-)Konfiguration ermitteln, welchen Wertebereich ein Feature-Attribut annehmen kann. Diese Analyse nennen sie *Attribute Range Analysis*. Die Ergebnisse dieser Analyse sind ähnlich zu den in Kapitel 3 beschriebenen Aggregationsfunktionen für Minimum und Maximum. Wie in dieser Arbeit kodieren sie die Anfragen für SMT-Solver, um deren Funktionalitäten für arithmetische Operationen verwenden zu können.

Ein ähnliches Vorgehen erläutern Zanardini et al. [2016]. Sie ermitteln für eine Auswahl an Features für eine vom Nutzer gewählte Zielfunktion ein Set an validen Konfigurationen. Dabei untersuchen sie für jedes im Feature-Modell enthaltene Feature durch Code-Analysen und Heuristiken, welchen Ressourcenverbrauch dieses Feature besitzt und fügen entsprechende Anmerkungen an dieses Feature im Feature-Modell an. Diese Anmerkungen verwenden sie dann für die Suche von validen Konfigurationen auf Basis der bisherigen Featureauswahl des Nutzers.

7. Fazit

Beim Konfigurieren einer Software-Produktlinie haben Nutzer derzeit nicht die Möglichkeit, eigene Constraints über den im Feature-Modell enthaltenen Feature-Attributen zusätzlich zu den Constraints im Feature-Modell zu definieren, um spezielle Anforderungen an die zu erstellende Variante in den Konfigurationsprozess einfließen zu lassen. Qualitative Anforderungen, wie dass die Variante nur einen maximalen Ressourcenverbrauch aufweisen darf, werden somit nicht bei der Konfiguration berücksichtigt.

Unsere Arbeit trägt ein Konzept bei, mit dem zusätzliche Constraints über Feature-Attributen zu formulieren und stellt ein Konzept vor, diese Constraints für die Verwendung mit SMT-Solvern in SMT-Syntax umzuwandeln.

In Kapitel 3 stellten wir eine Klassifizierung von Anwendungsfällen vor, die die Anwendungsfälle aufgrund der Zeitpunkte definiert, wann Feature-Attributen im SPL-Entwicklungsprozess zugewiesen werden und wann die Constraints definiert werden. Feature-Attribute klassifizierten wir als Model-Attribute beziehungsweise Configuration-Attribute, während wir Constraints in Model-Constraints und Configuration-Constraints einteilten.

Während der Klassifizierung der Anwendugnsfälle stellten wir anhand von Beispielen fest, dass die Anwendungsfälle sich nicht immer klar voneinander abgrenzen lassen. Vielmehr treten Kombinationen von Anwendungsfällen auf. Weiterhin definierten wir in Kapitel 3 eine Syntax für Feature-Attribute und die Aggregationsfunktionen Summe, Produkt, arithmetisches Mittel, Maximum und Minimum. Darauf aufbauend beschrieben die Überführung dieser Formeln in SMT-Syntax.

Das Konzept wandten wir in Kapitel 4 an, um einen Prototypen für die Evaluierung zu implementieren. Dabei verwendeten wir ein Java-Framework auf Basis von JavaSMT, um die im Konzept vorgestellte Syntax anzuwenden. In unseren Ausführungen gingen wir auf den Prozess des Modell-Einlesens und der Kodierung mittels JavaSMT ein. Dabei stellten wir fest, dass nicht alle von JavaSMT unterstützten Solver für die Umsetzung unseres Konzeptes geeignet sind.

Bei der Evaluierung in Kapitel 5 führten wir einige Experimente für vier realistische Modelle unterschiedlicher Größenordnungen durch. Dabei stellten wir fest, dass die Anwendung des Solvers eine höhere Laufzeit benötigte, je mehr Features im Feature-Modell enthalten sind. Bei der Überprüfung, ob sich Aggregationsfunktionen auf die Laufzeit des Solvers auswirken ließen, sich keine klaren Erkenntnisse ziehen.

Zukünftige Arbeiten

Durch die Implementierung des Konzept mithilfe eines Prototypen und die daraus resultierende Evaluierung haben sich weitere Themenstellungen gezeigt, die Gegenstand zukünftiger Arbeiten werden können. So wurden die Constraints über den Feature-Attributen bei der Implementierung in SMT-Syntax umgesetzt. Da SAT-Solver häufig effizienter sidn [Mendonca et al., 2009], könnte untersucht werden, welche Schritte für die Überführung der Constraints in aussagenlogische Formeln vonnöten sind.

Weiterhin wurden die Ergebnisse dieser Arbeit durch die Verwendung eines einzelnen SMT-Solvers ermittelt. Da JavaSMT weitere SMT-Solver unterstützt, können in einer zukünftigen Arbeit unterschiedliche SMT-Solver in Bezug auf Unterstützung und Laufzeit der Analysen miteinander verglichen werden.

In Abschnitt 4.1 beschreiben wir, dass wir alle möglichen Variablen und Formeln für die Feature-Attribute beim Einlesen des Feature-Modell definieren. In einer nachfolgenden Arbeit kann daher untersucht werden, ob und wie es möglich ist, nur die Variablen und Formeln für die Feature-Attribute zu definieren, die durch Aggregationsfunktionen notwendig sind.

Durch die Evaluierung in Abschnitt 5.3 stellen wir fest, dass für größere Modelle die Laufzeit zunimmt. Weiterführende Arbeiten können daher untersuchen, wo die Grenzen der Umwandlung in die SMT-Syntax liegen.

Anhang


```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<extendedFeatureModel>
  <struct>
    <and mandatory="true" name="Webserver">
      <graphics key="collapsed" value="false"/>
      <attribute name="Latenz" type="long" unit="ms" value="10"/>
      <attribute name="MAnf" type="long" unit="" value="1000000"/>
      <and mandatory="true" name="Protokoll">
        <feature name="HTTP"/>
        <feature name="HTTPS"/>
        <feature name="File"/>
      </and>
      <or name="Scriptsprachen">
        <attribute name="Latenz" type="long" unit="ms" value="20"/>
        <attribute name="MAnf" type="long" unit="" value="25000"/>
        <feature name="PHP"/>
        <feature name="JSP"/>
      </or>
      <alt name="Datenbank">
        <graphics key="collapsed" value="false"/>
        <attribute name="Latenz" type="long" unit="ms" value="20"/>
        <feature name="MySQL">
          <attribute name="MAnf" type="long" unit="" value="300000"/>
        </feature>
        <feature name="PostgreSQL">
          <attribute name="MAnf" type="long" unit="" value="50000"/>
        </feature>
      </alt>
      <feature name="SSL">
        <attribute name="Latenz" type="long" unit="ms" value="10"/>
        <attribute name="MAnf" type="long" unit="" value="500000"/>
      </feature>
      <and mandatory="true" name="Generierung">
        <graphics key="collapsed" value="false"/>
        <feature name="Dynamisch"/>
        <and name="Statisch">
          <feature name="Dateisystem">
```

```
          <attribute name="Latenz" type="long" unit="ms" value="20"/
            >
          <attribute name="MAnf" type="long" unit="" value="20000"/>
        </feature>
        <feature name="WAR">
          <attribute name="Latenz" type="long" unit="ms" value="20"/
            >
          <attribute name="MAnf" type="long" unit="" value="400000"/
            >
        </feature>
      </and>
    </and>
  </and>
</struct>
<constraints>
  <rule>
    <imp>
      <not>
        <disj>
          <var>HTTP</var>
          <var>HTTPS</var>
        </disj>
      </not>
      <var>File</var>
    </imp>
  </rule>
  <rule>
    <imp>
      <var>File</var>
      <var>Dateisystem</var>
    </imp>
  </rule>
</constraints>
</extendedFeatureModel>
```

Quelltext A.1: Feature-Modell aus Abbildung 2.2 im XML-Format von FeatureIDE

Literaturverzeichnis

Andreas Abele, Yiannis Papadopoulos, David Servat, Martin Törngren, und Matthias Weber. The CVM Framework-A Prototype Tool for Compositional Variability Management. *VaMoS*, pages 101–105, 2010. (zitiert auf Seite 1)

Mathieu Acher, Philippe Collet, Philippe Lahire, und Robert B France. FAMILIAR: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78:657–681, 2013. doi: 10.1016/j.scico.2012.12.004. (zitiert auf Seite 1)

Mauricio Alférez, Mathieu Acher, José A Galindo, Benoit Baudry, und David Benavides. Modeling variability in the video domain: language and experience report. *Software Quality Journal*, 27:307–347, 2019. (zitiert auf Seite 1)

Sven Apel und Christian Kästner. An overview of feature-oriented software development. *J. Object Technol.*, 8:49–84, 2009. (zitiert auf Seite 14)

Sven Apel, Don Batory, Christian Kästner, und Gunter Saake. Software product lines. In *Feature-Oriented Software Product Lines*, pages 3–15. Springer, 2013. (zitiert auf Seite 1, 2, 5, 6, 7, 8, 11, 12, 13, 14 und 16)

Al Azzawi und Ali Fouad. PyFml-a Textual Language For Feature Modeling. *International Journal of Software Engineering & Applications (IJSEA)*, 9, 2018. doi: 10.5121/ijsea.2018.9104. (zitiert auf Seite 1)

Clark Barrett, Pascal Fontaine, und Cesare Tinelli. The SMT-LIB Standard Version 2.6. 2015. (zitiert auf Seite xi, 2, 14, 26, 27, 36, 38 und 39)

Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005. doi: 10.1007/11554844_3. (zitiert auf Seite 1)

David Benavides, Pablo Trinidad, und Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. *International Conference on Advanced Information Systems Engineering*, pages 491–503, 2005. (zitiert auf Seite 1, 2, 9, 11, 17, 59 und 60)

David Benavides, Sergio Segura, Pablo Trinidad, und Antonio Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. 2007. (zitiert auf Seite 2)

David Benavides, Sergio Segura, und Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information systems*, 35: 615–636, 2010. (zitiert auf Seite 1 und 9)

Daniel Le Berre und Anne Parrain. The Sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. (zitiert auf Seite 14)

Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, und Rudolf Schlatte. Variability modelling in the abs language. In *International Symposium on Formal Methods for Components and Objects*, pages 204–224. Springer, 2010. doi: 10.1007/978-3-642-25271-6_11. (zitiert auf Seite 1)

Andreas Classen, Patrick Heymans, und Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. 2008. (zitiert auf Seite 6)

Andreas Classen, Quentin Boucher, und Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76:1130–1143, 2011. doi: 10.1016/J.SCICO.2010.10.005. (zitiert auf Seite 1)

Paul Clements. *Software Product Lines*. Addison-Wesley Boston, 1999. (zitiert auf Seite 6)

Deepak Dhungana, Paul Grünbacher, und Rick Rabiser. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering 2010 18:1*, 18:77–114, 2010. doi: 10.1007/S10515-010-0076-6. (zitiert auf Seite 1)

Niklas Eén und Niklas Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006. (zitiert auf Seite 1)

Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, und Alexander Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. pages 391–400, 2014. (zitiert auf Seite 1 und 5)

Sebastian Henneberg. Next-generation feature models with pseudo-boolean sat solvers. *Bachelor thesis, Department of Informatics and Mathematics, University of Passau*, page 36, 2011. (zitiert auf Seite 59)

Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, und Andrzej Wasowski. Clafer: Lightweight modeling of structure, behaviour, and variability. *Art Sci. Eng. Program.*, 3:2, 2019. (zitiert auf Seite 1)

Ahmet Serkan Karataş, Halit Oğuztüzün, und Ali Doğru. From extended feature models to constraint logic programming. *Science of Computer Programming*, 78: 2295–2312, 2013. (zitiert auf Seite 59)

Peter Knauber, Jesus Bermejo, Günther Böckle, Julio Cesar Sampaio do Prado Leite, Frank van der Linden, Linda Northrop, Michael Stark, und David M Weiss. Quantifying Product Line Benefits. pages 155–163, 2001. doi: 10.1007/3-540-47833-7_15. (zitiert auf Seite 5 und 6)

Sebastian Krieter, Thomas Thüm, Reimar Schröter, Gunter Saake, und Sandro Schulze. Propagating Configuration Decisions with Modal Implication Graphs. page 12, 2018. doi: 10.1145/3180155.3180159. (zitiert auf Seite 12)

Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, und Gunter Saake. Extracting Software Product Lines: A Cost Estimation Perspective. pages 354–361. Association for Computing Machinery, 2016. doi: 10.1145/2934466.2962731. (zitiert auf Seite 5)

Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, und Sven Apel. FeatureIDE: A tool framework for feature-oriented software development. pages 611–614, 2009. doi: 10.1109/ICSE.2009.5070568. (zitiert auf Seite 1)

Marcilio Mendonca, Moises Branco, und Donald Cowan. S.P.L.O.T. - Software product lines online tools. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pages 761–762, 2009. doi: 10.1145/1639950.1640002. (zitiert auf Seite 1 und 62)

Sonia Montagud, Silvia Abrahão, und Emilio Insfran. A systematic review of quality attributes and measures for software product lines. *Software Quality Journal*, 20: 425–486, 2012. doi: 10.1007/s11219-011-9146-7. (zitiert auf Seite 10)

Leonardo De Moura und Nikolaj Bjørner. LNCS 4963 - Z3: An Efficient SMT Solver. 2008. doi: https://doi.org/10.1007/978-3-540-78800-3_24. (zitiert auf Seite 14 und 45)

Lina Ochoa, Oscar González-Rojas, und Thomas Thüm. Using Decision Rules for Solving Conflicts in Extended Feature Models. *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 149–160, 2015. doi: 10.1145/2814251.2814263. (zitiert auf Seite 10)

Klaus Pohl, Günter Böckle, und Frank Van Der Linden. *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer, 2005. (zitiert auf Seite 1)

Klaus Schmid und Martin Verlage. The economic impact of product line adoption and evolution. *IEEE software 19.4*, pages 50–57, 2002. (zitiert auf Seite 5)

Klaus Schmid, Christian Kröher, und Sascha El-Sharkawy. Variability modeling with the integrated variability modeling language (IVML) and EASy-producer. *ACM International Conference Proceeding Series*, 1:306, 2018. doi: 10.1145/3233027.3233057. (zitiert auf Seite 1)

Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, und Ina Schaefer. SMT-Based Variability Analyses in FeatureIDE. 2020. doi: 10.1145/3377024.3377036. (zitiert auf Seite 60)

Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, und Thomas Thüm. Yet Another Textual Variability Language? A Community Effort Towards a Unified Language. 2021. doi: 10.1145/3461001.3471145. (zitiert auf Seite 1)

Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, und Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14:254–272, 2009. doi: 10.1007/s10601-008-9061-0. (zitiert auf Seite 2)

Reinhard Tartler, Daniel Lohmann, Julio Sincero, und Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. *Proceedings of the sixth conference on Computer systems*, pages 47–60, 2011. (zitiert auf Seite 11)

Andrzej Wasowski, Krzysztof Czarnecki, Marcilio Mendonca, und Andrzej W˛ Asowski. SAT-based analysis of feature models is easy. *Proceedings of the 13th International Software Product Line Conference*, pages 231–240, 2009. doi: 10.1145/1753235.1753267. (zitiert auf Seite 14)

Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, und Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012. (zitiert auf Seite 56)

Damiano Zanardini, Elvira Albert, und Karina Villela. Resource–usage–aware configuration in software product lines. *Journal of Logical and Algebraic Methods in Programming*, 85:173–199, 2016. (zitiert auf Seite 11, 16, 17 und 60)

Guoheng Zhang, Huilin Ye, und Yuqing Lin. Using Knowledge-Based Systems to Manage Quality Attributes in Software Product Lines. *Proceedings of the 15th International Software Product Line Conference*, pages 1–7, 2011. doi: 10.00. (zitiert auf Seite 10)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Weiterhin wurde die Arbeit bisher weder in einem anderen Prüfungsverfahren vorgelegt noch anderweitig veröffentlicht. Textpassagen, die wörtlich oder dem Sinn nach auf anderen Quellen beruhen, sind als solche kenntlich gemacht.

Magdeburg, 2. Mai 2022