Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Masterarbeit

Effiziente Kodierung von Variabilität in Spezifikationen

Autor:

Matthias Praast

24. Februar, 2014

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Inform. Thomas Thüm
M.Sc. Reimar Schröter
Institut für Technische und Betriebliche Informationssysteme (ITI)

Praast, Matthias:
Effiziente Kodierung von Variabilität in Spezifikationen
Masterarbeit, Otto-von-Guericke-Universität Magdeburg, 2014.

Inhaltsangabe

Software-Produktlinien beschreiben eine Menge ähnlicher Software-Produkte mit einer gemeinsamen Code-Basis. Software-Produktlinien können mittels Feature-orientierter Programmierung entwickelt werden. Aufgrund der Gemeinsamkeiten und Unterschiede der einzelnen Software-Produkte werden Features definiert. Um ein spezielles Produkt zu erzeugen, werden die benötigten Features komponiert. Innerhalb der Features können Methoden mithilfe von Kontrakten spezifiziert werden. Um diese Spezifikationen bei der Verifikation eines Software-Produktes nutzen zu können, müssen auch die Kontrakte komponiert werden. Für die Komposition von Kontrakten wurden verschiedene Verfahren vorgeschlagen.

Die Verifikation einer kompletten Software-Produktlinie ist schwierig, da die Anzahl der möglichen Produkte bis zu exponentiell in der Anzahl der Features steigt. Um nicht jedes Software-Produkt einzeln verifizieren zu müssen, wurde vorgeschlagen, die Auswahl der Features mithilfe eines Metaproduktes in die Laufzeit zu überführen. Um auch Spezifikationen bei der Verifikation nutzen zu können, müssen diese ebenfalls die Feature-Auswahl zur Laufzeit einbeziehen.

In dieser Arbeit wird eine vorhandene Erzeugung eines Metaproduktes um weitere Kompositionsverfahren für Spezifikationen erweitert. Die so erzeugten Spezifikationen lassen sich vereinfachen. Die für die Vereinfachungen nötigen Regeln werden in dieser Arbeit vorgestellt. Es wird untersucht, welche Auswirkungen die Optimierungen für die Verifikation mithilfe eines *Model Checkers* beziehungsweise eines *Theorem Provers* haben. Außerdem wird der Einfluss der Optimierungen auf die Lesbarkeit der Klauseln betrachtet.

Danksagung

Hiermit möchte ich mich bei meinen Betreuern Reimar Schröter, Thomas Thüm und Prof. Gunter Saake bedanken, dass sie mir die Möglichkeit gegeben haben, diese Masterarbeit zu schreiben. Ohne die konstruktive Kritik von und die Diskussionen mit Thomas und Reimar wäre die Arbeit in diesem Umfang nicht möglich gewesen.

Ein Dank geht auch an Fabian Benduhn und Jens Meinicke für die Unterstützung bei der Einrichtung von FeatureIDE, MonKeY und Java Pathfinder.

Bei Prof. Frank Ortmeier möchte ich mich für die hilfreichen Anmerkungen zu den Ergebnissen der Verifikation mit MonKeY bedanken.

Als letztes möchte ich mich bei meiner Familie bedanken, die mich während der gesamten Studienzeit unterstützt hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

1. Einführung

Die heutigen Software-Systeme werden immer umfangreicher. Bei der Entwicklung von ähnlichen Software-Systemen reicht die Objekt-Orientierung nicht immer aus, um alle benötigten Varianten abzubilden. Häufig werden dann Software-Varianten kopiert und angepasst (engl. clone and own bzw. copy and modify [ABKS13]). Dies hat jedoch den Nachteil, dass dabei auch Fehler mit kopiert werden. Diese müssen dann in allen Kopien behoben werden. Weiterhin findet die Weiterentwicklung in der Regel nicht in allen Varianten parallel statt, sodass der Programmcode in den verschiedenen Varianten schnell auseinander läuft [ABKS13].

Eine andere Möglichkeit der Wiederverwendung von Code bieten Software-Produktlinien. Bei Software-Produktlinien werden die Gemeinsamkeiten und Unterschiede der einzelnen Varianten durch Features abgebildet. Diese Features können separat implementiert und weiterentwickelt werden. Einzelne Software-Varianten können dann aus diesen Features generiert werden. Eine Möglichkeit der Implementierung von Software-Produktlinien ist die Feature-orientierte Programmierung. Dabei werden für jedes Feature sogenannte Feature-Module implementiert. Bei der Generierung von Software-Produkten werden diese Module dann komponiert [ABKS13].

Die Entwicklung von Feature-orientierten Software-Produktlinien hat jedoch auch einige Schwierigkeiten. Zum Einen muss sich jedes Software-Produkt kompilieren lassen. Zum Anderen sollen alle erzeugte Software-Produkte fehlerfrei arbeiten. Unterscheiden sich zwei Produkte beispielsweise nur darin, dass in einem Produkt ein Feature gewählt wurde, im anderen aber nicht, sind sämtliche Programmteile, die von diesem Feature nicht beeinflusst werden, identisch, müssen aber trotzdem zweimal kompiliert und verifiziert werden. Die Erzeugung aller Produkte kann außerdem sehr aufwendig sein [ABKS13]. Bereits bei nur 10 Features gibt es bis zu 1024 mögliche Produkte, die generiert und verifiziert werden müssen. Viele Berechnungen und Tests würden dabei mehrfach ausgeführt werden. Auch wenn zwei Produkte viele Gemeinsamkeiten im Coding haben, müssen diese bei beiden Produkten kompiliert und verifiziert werden. Die dafür nötigen Berechnungen sind jedoch die gleichen.

Um Software-Produktlinien zu verifizieren, muss deren Funktionalität zunächst spezifiziert werden. Eine mögliche Spezifikationstechnik stellt Design By Contract dar

[TSK+12]. Hierbei wird mithilfe von Vor- und Nachbedingungen sowie Invarianten das erwartete Verhalten von Methoden oder Programmen festgelegt. Eine Möglichkeit solche Spezifikationen für Java-Programme zu erstellen, bietet die Java Modeling Lanuage (JML) [LC03].

Um einzelne Software-Produkte einer Software-Produktlinie zu verifizieren, können unterschiedliche Methoden angewandt werden. Dass ein Programm seinen Spezifikationen entspricht, kann mithilfe von Theorem Proving nachgewiesen werden. Die Korrektheit wird dabei formal bewiesen [Ehr02]. Eine weitere Möglichkeit bietet das Model Checking. Dabei wird geprüft, ob die Ausführung des Programms zu einem Zustand führen kann, der nicht den Spezifikationen entspricht [DKW08].

Um die Verifikationsmethoden für einzelne Software-Produkte auf Software-Produktlinien anwenden zu können, haben Post und Sinz [PS08] sowie Apel et al. [ASW+11] die Erzeugung eines Metaproduktes vorgeschlagen. Dieses Metaprodukt repräsentiert alle möglichen Software-Produkte einer Software-Produktlinie. Dazu wird die Auswahl der Features in die Laufzeit überführt. Meinicke hat die Generierung dieses Metaproduktes implementiert [Mei13]. Anschließend wurden Tools für Model Checking und Theorem Proving auf das generierte Metaprodukt angewendet. Es wurde jedoch nicht geprüft, wie diese sich bei sehr großen und komplexen Software-Produktlinien verhalten.

Weiterhin existieren verschiedene Verfahren zur Komposition von Spezifikationen zweier Features. Meinicke hat jedoch lediglich das Verfahren *Explicit Contract Refinement* betrachtet.

Zielstellung der Arbeit

In dieser Arbeit wird betrachtet, wie die Komposition von Kontrakten im Metaprodukt mithilfe weiterer Kompositionsverfahren erfolgen kann. Dafür sollen die Kompositionsmechanismen *Conjunctive Contract Refinement*, *Consecutive Contract Refinement*, *Cumulative Contract Refinement*, *Plain Contracting* und *Method-based Contract Refinement* betrachtet werden. Weiterhin soll untersucht werden, wie sich die erzeugten Spezifikationen aufgrund von Redundanzen und Beziehungen zwischen Features vereinfachen lassen. Mithilfe der so aufgestellten Regeln sollen die Tools *FeatureIDE* [TKB+14] und *FeatureHouse* [AKL13] erweitert werden, sodass die Generierung eines Metaproduktes unter Verwendung der zusätzlichen Kompositionsverfahren möglich ist.

Ziel der Vereinfachungen der Spezifikationen ist es, die Effizienz der Verifikation zu steigern. Daher sollen die Auswirkungen einzelner Optimierungen auf die Verifikation mithilfe des *Theorem Provers MonKeY* und des *Model Checkers Java Pathfinder* betrachtet werden. Eine bessere Lesbarkeit erleichtert zum einen das Erkennen und Beheben eines Fehlers, zum anderen kann dadurch auch die Entwicklung beziehungsweise Weiterentwicklung der Software-Produktlinie erleichtert werden. Daher soll auch die Lesbarkeit der Spezifikationen untersucht werden. Da für die Optimierungen zusätzliche Berechnungen erforderlich sind, soll auch die Veränderung der Generierungszeit eines Metaproduktes evaluiert werden.

Gliederung der Arbeit

In Kapitel 2 werden zunächst die Grundlagen vorgestellt, die zum Verständnis der Arbeit nötig sind. In Kapitel 3 werden Regeln für die Erzeugung von Spezifikationen im Metaprodukt sowie zu deren Vereinfachung erläutert. Anschließend wird in Kapitel 4 die Implementierung dieser Regeln in den Tools *FeatureIDE* und *FeatureHouse* beschrieben. Im Kapitel 5 folgt die Betrachtung der Auswirkungen der Vereinfachungen. Es werden sowohl die Auswirkungen auf die Lesbarkeit, als auch auf die Verifikation anhand einer Software-Produktlinie beschrieben. Verwandte Arbeiten werden in Kapitel 6 vorgestellt. Eine Zusammenfassung der Arbeit folgt in Kapitel 7. Mögliche Themen für zukünftige Arbeiten werden in Kapitel 8 betrachtet.

2. Grundlagen

Um das Verständnis der Arbeit zu erleichtern werden in diesem Kapitel zunächst die Grundlagen dargelegt. In Abschnitt 2.1 wird zunächst die Feature-orientierte Entwicklung von Software-Produktlinien erläutert. Eine Möglichkeit Funktionalitäten zu spezifizieren bietet Design By Contract. Dies wird in Abschnitt 2.2 dargestellt. In Abschnitt 2.3 wird beschrieben, wie die so erstellten Spezifikationen für die Verifikation genutzt werden können. Anschließend wird in Abschnitt 2.4 erläutert wie die in Abschnitt 2.2 beschriebene Spezifikationstechnik für Software-Produktlinien angewendet werden kann. In Abschnitt 2.5 wird erklärt, wie Software-Produktlinien verifiziert werden können.

2.1 Software-Produktlinien

Häufig werden ähnliche Software-Produkte immer wieder neu entwickelt. Oft kann objektorientierte Programmierung genutzt werden, um einzelne Funktionalitäten in mehreren Software-Produkten nutzen zu können. Dies ist jedoch nicht immer ausreichend.

Software-Produktlinien stellen eine Möglichkeit der massenweisen Erstellung und Anpassung von Software-Produkten dar [ABKS13]. Sie ermöglichen die Individualisierung von Software-Produkten, ermöglichen aber gleichzeitig die Massenproduktion in dem gesamten Anwendungsgebiet und Marktsegment. Hierdurch können Entwicklungskosten gespart werden [CN06].

In Abbildung 2.1 auf der nächsten Seite sind die Kosten für die Entwicklung einer Software-Produktlinie den Kosten für die Entwicklung einzelner Software-Produkte gegenüber gestellt. Zunächst sind die Kosten höher als bei der Entwicklung eines einzelnen Software-Produktes, da zunächst analysiert werden muss, welche Programmvarianten möglich sein sollen und welche Gemeinsamkeiten und Unterschiede diese haben. Die Generierung einzelner Software-Produkte aus der so erstellten Code-Basis ist dann jedoch mit deutlich weniger Aufwand verbunden, als die Erstellung eines einzelnen Produktes. Bei drei Produkten, sind die Kosten für die Entwicklung einer Software-Produktlinie und die separate Entwicklung dreier Software-Produkte etwa

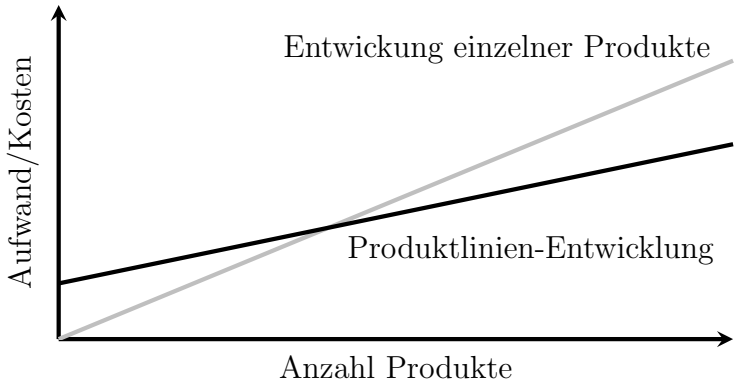

Abbildung 2.1: Vergleich der Kosten für die Entwicklung einer Software-Produktlinie mit der Entwicklung aller Software-Produkte Quelle:[ABKS13]

gleich hoch. Die Anzahl an Produkten, ab denen die Entwicklung einer Software-Produktlinie günstiger ist, als die Entwicklung einzelner Software-Produkte wird in der Literatur mit drei bis vier angegeben [CN06, PBvdL05].

2.1.1 Feature Modelle

Mithilfe von Software-Produktlinien lassen sich Software-Produkte erzeugen, die gemeinsame Eigenschaften haben. Diese Gemeinsamkeiten und Unterscheide lassen sich durch Features abbilden . Ein Feature ist eine, für den Anwender sichtbarere Funktionalität eines Software-Systems [KCH+90]. Ein Software-Produkt einer Software-Produktlinie wird aus einer Teilmenge aller Features der Software-Produktlinie erzeugt. Da nicht alle Features immer miteinander kombinierbar sind, beziehungsweise einige Features auch andere Features voraussetzen können, müssen zunächst die Abhängigkeiten zwischen den Features analysiert werden (engl. Domain Analysis) [KCH+90]. Das Ergebnis ist ein Feature Model. Das Feature Model beschreibt, die Features der Software-Produktlinie und deren Beziehungen untereinander [ABKS13]. In Feature Modellen werden zwei Arten von Features unterschieden: konkrete und abstrakte Features [TKES11]. Während konkrete Features sich direkt mit Implementierungen verknüpfen lassen, dienen abstrakte Features lediglich der Gruppierung und Strukturierung der Features.

Feature-Modelle können auf verschiedene Weisen repräsentiert werden. Eine formale Möglichkeit bieten aussagenlogische Ausdrücke. Grafisch lassen sich Feature-Modelle mithilfe von Feature-Diagrammen darstellen. Feature-Diagramme sind Bäume deren Knoten einzelne Features repräsentieren [ABKS13]. Features, die Kinder eines anderen Features sind, können nur gewählt werden, wenn auch das Eltern-Feature gewählt wurde. Einzelne Kinder eines Features können optional oder obligatorisch gewählt werden. Obligatorische Features sind in allen Software-Produkte enthalten, die auch das Eltern-Feature enthalten. Bei optionalen Features muss dies nicht der Fall sein. Ein Kind-Feature kann ein Teil einer Oder-Gruppe oder einer Alternativen-Gruppe sein, wenn das Eltern-Feature gewählt wurde. Aus einer Oder-Gruppe muss mindestens ein Feature gewählt werden, aus einer Alternativen-Gruppe genau eines, vorausgesetzt, das Eltern-Feature wurde gewählt. Da dies häufig noch nicht ausreichend ist, weil Features auf unterschiedlichen Ästen des Baumes sich gegenseitig

ausschließen oder voraussetzen könnten, können auch zusätzliche, logische Formeln, sogenannte Constraints angegeben werden.

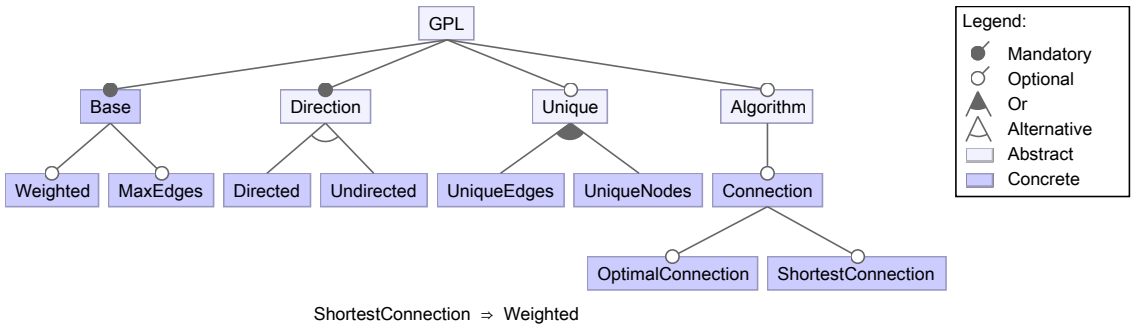

Abbildung 2.2: Feature-Modell einer Software-Produktlinie für einen Graphen

In Abbildung 2.2 ist beispielhaft ein Feature-Modell für das von Lopez-Herrejon und Batory vorgeschlagene Standard-Problem für Software-Produktlinien dargestellt [LHB01]. Es handelt sich hierbei um eine Software-Produktlinie für die Repräsentation von Graphen. Diese Software-Produktlinie enthält allerdings nicht alle vorgeschlagenen Features. Die Software-Produktlinie wurde von Weigelt erstellt [Wei13] und ist als Beispiel im Eclipse-Plugin *FeatureIDE* [TKB+14] enthalten.

Das Abstrakte Feature *GPLscratchFHJML* als Wurzelknoten des Diagramms muss immer ausgewählt sein. *Base* und *Direction* sind obligatorische Features, müssen also in jedem Software-Produkt vorhanden sein. In Abbildung 2.2 sind diese als *Mandatory* gekennzeichnet. Während jedoch *Base* ein konkretes Feature ist und direkten Einfluss auf den Quellcode hat, sind die restlichen Kinder-Features von *GPLscratchFHJML* abstrakte Features und dienen lediglich dazu, weitere Features zu gruppieren.

Mithilfe der Features *Weighted* und *MaxEdges* können dem Graphen zusätzliche Eigenschaften hinzugefügt werden. Mithilfe des Features *Weightes* werden Kantengewichte hinzugefügt. Das Feature *MaxEdges* dient dazu, eine maximale Anzahl an Kanten für einen Graphen zu definieren. Beides sind Kinder-Features des Features *Base* und können daher nur dann ausgewählt werden, wenn auch das Feature *Base* gewählt wurde. Da *Base* aber ein obligatorisches Kind-Feature des Wurzelfeatures ist, können *Weighted* und *MaxEdges* immer ausgewählt werden.

Jeder Graph ist entweder gerichtet oder ungerichtet. Dies wird durch das abstrakte Feature *Direction* mit seinen Kinder-Features *Directed* und *Undirected* deutlich. Die beiden Kinder-Features stellen eine Alternative dar. Da *Direction* obligatorisch ist, muss auch immer eines der beiden Kinder-Features in jedem Software-Produkt vorhanden sein - jeder Graph ist entweder gerichtet oder ungerichtet, niemals beides oder keines von beidem.

Mit dem abstrakten Feature *Unique* können Kanten oder Knoten eindeutig im Graphen identifiziert werden. *Unique* dient dabei lediglich der Gruppierung. Da die beiden Kinder-Features *UniqueEdges* und *UniqueNodes* eine Oder-Gruppe bilden, muss mindestens eines der beiden Features gewählt werden, wenn *Unique* gewählt wird. *UniqueEdges* dient dabei der eindeutigen Identifizierung von Kanten, *UniqueNodes* dient der eindeutigen Identifizierung von Knoten.

Das abstrakte Feature *Algorithm* fasst alle Algorithmen zusammen, die für den Graphen durchlaufen werden können. Mithilfe des Kind-Features *Connection* kann ermittelt werden, ob eine Verbindung zwischen zwei Knoten existiert. Die Kind-Features von *Connection* stellen Spezialisierungen der Berechnung dar. Sie können nur vorhanden sein, wenn auch das Feature *Connection* vorhanden ist. Mithilfe des Features *OptimalConnection* wird die Berechnung der Verbindung mit den wenigsten Knoten ermöglicht, mit dem Feature *ShortestConnection* ist die Berechnung der Verbindung mit den niedrigsten Kantengewichten möglich.

Die Berechnung der kürzesten Verbindung, die durch das Feature *ShortestConnection* ermöglicht wird, ist nur möglich, wenn der Graph Kantengewichte enthält. Daher setzt das Feature *ShortestConnection* das Feature *Weighted* voraus. Dies wird durch den unter dem Graphen dargestellten Constraint $ShortestConnection \Rightarrow Weighted$ abgebildet.

2.1.2 Feature-orientierte Programmierung

Eine Menge von ausgewählten Features wird auch als Konfiguration bezeichnet. Unter Nutzung der Konfiguration wird ein Software-Produkt einer Software-Produktlinie generiert. Um ein valides Software-Produkt zu erhalten, muss die Konfiguration bezogen auf das Feature-Modell gültig sein.

Um anhand einer Konfiguration Software-Produkte zu generieren können verschiedene Verfahren angewendet werden. Eines dieser Verfahren ist die Feature-orientierte Programmierung. Die Feature-orientierte Programmierung kann als Generalisierung der Objekt-orientierten Programmierung betrachtet werden [Pre97]. Neben Objekt-orientierten Sprachen kann die Feature-orientierte Programmierung auch für andere Programmierparadigmen, wie XML-basierte Sprachen oder funktionale Programmierung angepasst werden [AKL13].

Mathematisch lässt sich die Kombination zweier Features aus der Feature-Menge F mithilfe des Operators $\bullet$ folgendermaßen beschreiben:

$$\bullet : F \times F \rightarrow F \tag{2.1}$$

Zwei Features werden kombiniert. Das Ergebnis ist wiederum ein Feature mit den Eigenschaften der beiden ursprünglichen Features. Wenn die Features widersprüchliche Eigenschaften haben, kann das neue Feature nur eine der Eigenschaften annehmen. Daher ist die Reihenfolge der Komposition relevant. Die Komposition ist nicht kommutativ. Um ein Software-Produkt p aus einer Menge von n gewählten Features $f_1, f_2, ...f_n$ zu generieren wird diese Kombination mehrfach ausgeführt:

$$p = f_1 \bullet f_2 \bullet ... \bullet f_n \tag{2.2}$$

Bei der feature-orientierten Programmierung existiert zu jedem konkreten Feature ein Feature-Modul. Diese Feature-Module enthalten die für das Feature nötigen Software-Artefakte. Software-Artefakte sind alle Arten von Informationen, die Teil der

Software sind oder mit der Software in Verbindung stehen [AKL13]. Diese Informationen können sowohl Code-Teile (Pakete, Klassen, Methoden, ...) als auch Dokumente (Modelle, Dokumentation, ...) sein [AKL13]. Ein verbreiteter Ansatz, um solche Software-Artefakte zu kombinieren ist *Superimposition* [AL08]. Bei der *Superimposition* werden Software-Artefakte kombiniert, indem deren Unterstrukturen kombiniert werden [AL08]. Solche Unterstrukturen können zum Beispiel Pakete oder Klassen sein. Wenn beispielsweise zwei Software-Artefakte mit der gleichen Klasse kombiniert werden, werden die beiden Klassen fusioniert. Wenn beide Klassen Methoden oder Attribute mit dem gleichen Namen und den gleichen Parametern enthalten, müssen diese ebenfalls fusioniert werden. Das Ergebnis ist eine Klasse mit den Attributen und Methoden der beiden Ausgangsklassen und dem gleichen Namen, wie die ursprünglichen Klassen.

Ein Framework für die Kombination von Software mittels *Superimposition* liefern Apel et al. mit *FeatureHouse* [AKL13]. Die Software-Artefakte werden in einer hierarchischen Struktur abgebildet, sogenannten *Feature Structure Trees*. Mit dieser Struktur ist es möglich, ein Software-Produkt entsprechend der gewählten Features zu generieren.

Anhand des Beispiels aus Abbildung 2.2 auf Seite 7 soll die Funktionsweise von *FeatureHouse* und *Superimposition* näher erläutert werden. Abbildung 2.3 stellt vereinfachte Feature Structure Trees für die Features *Base* und *Weighted* dar. Beide Features beinhalten die Klasse *Edge*, für die Kanten zwischen zwei Konten. Das Feature *Basis* enthält zusätzlich die Klassen *Graph* für den kompletten Graphen und *Node* für die einzelnen Knoten des Graphen. Aus Gründen der Übersichtlichkeit, wurden die in Abbildung 2.3 die beiden Klassen *Graph* und *Node* nicht expandiert.

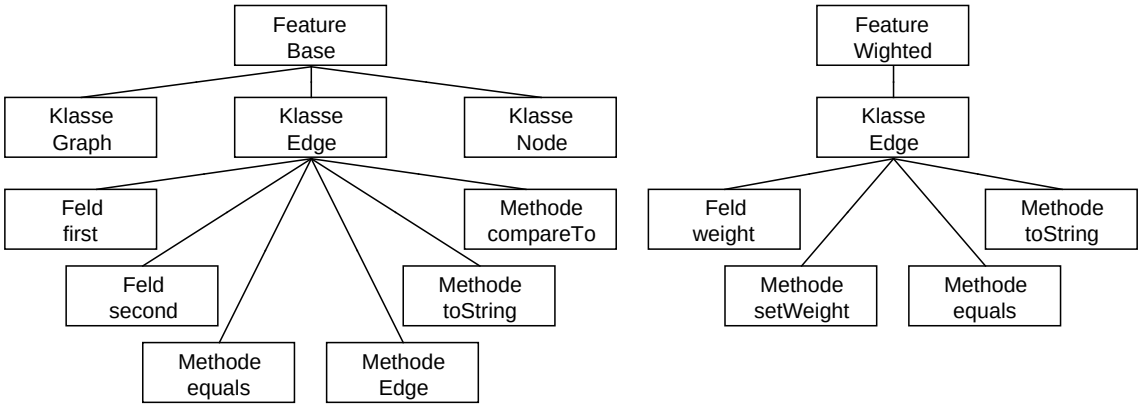

Abbildung 2.3: Feature Structure Trees für die Features *Base* und *Weighted*

In Abbildung 2.4 auf der nächsten Seite ist der Feature Structure Tree nach der Komposition der beiden Features abgebildet. Es sind alle drei Klassen vorhanden. Die neue Klasse *Edge* enthält die Methoden und Felder der *Edge*-Klassen aus beiden Features.

Bei der Komposition der beiden Features mit FeatureHouse werden die beiden Feature-Module kombiniert. Da die beiden Klassen *Graph* und *Node* lediglich im Feature *Base* enthalten sind, werden diese ohne Veränderung in das kombinierte

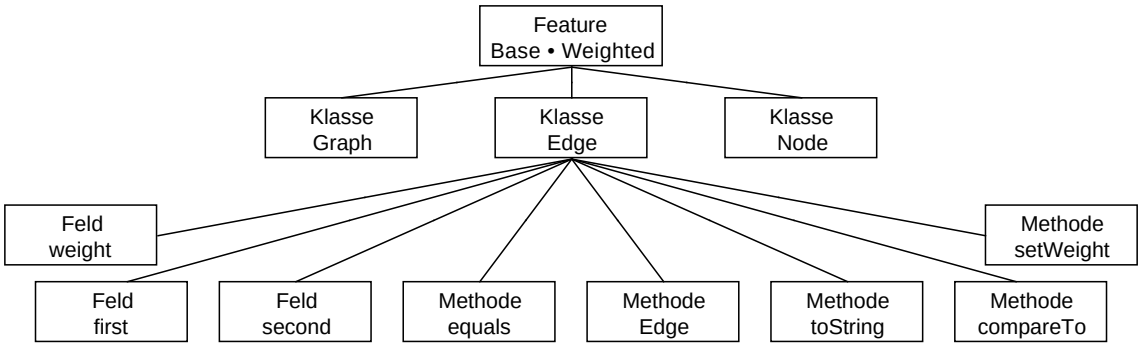

Abbildung 2.4: Feature Structure Tree für das Feature *Base • Weighted*

Feature übernommen. Weil die Klasse *Edge* in beiden Features vorhanden ist, müssen die beiden Implementierungen fusioniert werden.

In Quelltext 2.1 auf der nächsten Seite sind die Implementierungen der Klasse *Edge* in den Feature-Modulen beider Features dargestellt. Beide Klassen enthalten einen Konstruktor sowie die aus den Feature Structure Trees ersichtlichen Methoden und Attribute. Mit dem Aufruf von *original* in der Methode *equals* des Features *Weighted* wird gekennzeichnet, dass die ursprüngliche Methode aufgerufen wird.

```
1  public class Edge implements Comparable<Edge> {                    Base
2      private Node first;
3      private Node last;
4      public Edge(Node node1, Node node2){
5          // Konstruktor Edge(Node,Node) (Feature Base)
6      }
7      public boolean equals(Object obj){
8          return true;     // Aequivalenzberechnung (Feature Base)
9      }
10     public String toString(){
11         return "";       // String-Ausgabe (Feature Base)
12     }
13     public int compareTo(Edge edge){
14         return 0;        // Vergleich (Feature Base)
15     }
16 }
```

Das Vorgehen bei der Fusionierung zweier Klassen ähnelt der Komposition von zwei Feature-Modulen. Methoden und Felder, die in nur einer Klasse vorhanden sind, werden ohne Änderung in das komponierte Feature-Modul aufgenommen. Bei den Features *Base* und *Weighted* sind dies die Felder *first* und *last* aus der Klassen-Implementierung des Features *Base* und *weight* aus dem Feature *Weighted*, sowie die Methode *compareTo* aus dem Feature *Base* und *setWeight* aus dem Feature *Weighted*. Werden nur diese Felder und Methoden betrachtet, ergibt sich der in Quelltext 2.2 auf der nächsten Seite dargestellte Klassen-Quellcode.

Ist ein Feld in beiden Klassenimplementierungen enthalten, überschreibt die Implementierung im zweiten Feature die des ersten Features. Dies hat vor allem Auswirkungen auf die Initialisierung der einzelnen Felder.

```
1  public class Edge {                                                    Weighted
2      private int weight;
3      public Edge(Node node1, Node node2){
4          // Konstruktor Edge(Node,Node) (Feature Weighted)
5      }
6      public boolean equals(Object obj){
7          original(obj);
8          return true;    // Aequivalenzberechnung (Feature Weighted)
9      }
10     public String toString(){
11         return "";      // String-Ausgabe (Feature Weighted)
12     }
13     public void setWeight(int wgt){
14         // Kantengewicht setzen (Feature Weighted)
15     }
16 }
```

Quelltext 2.1: Klasse *Edge* in den Features *Base* und *Weighted*

```
1  public class Edge implements Comparable<Edge> {              Base • Weighted
2      private Node first;
3      private Node last;
4
5      public int compareTo(Edge edge){
6          // Vergleich (Feature Base)
7          return 0;
8      }
9
10     private int weight;
11
12     public void setWeight(int wgt){
13         // Kantengewicht setzen (Feature Weighted)
14     }
15 }
```

Quelltext 2.2: Klasse *Edge* des Feature-Moduls zu *Base* • *Weighted* ohne gemeinsame Felder und Methoden

Methoden können jedoch nicht einfach überschrieben werden. Statt dessen kann über den Aufruf von *original* der ursprüngliche Quellcode ausgeführt werden. In die Klasse des kombinierten Feature-Moduls wird dann zunächst die Methode des ersten Features eingefügt und umbenannt. In *FeatureHouse* wird dazu zunächst *__wrappee__* eingefügt, um zu kennzeichnen, dass es sich um eine umbenannte Methode handelt. Anschließend wird der Feature-Name angehängt, da die Methode durch mehrere Features überschrieben werden könnte. Danach wird die Methode des zweiten Features in die kombinierte Klasse aufgenommen. In dieser Methode wird der Aufruf von *original* durch den neuen Methoden-Namen ersetzt. Wenn in der Implementierung des zweiten Features kein Aufruf der Methode *original* vorhanden ist, kann die Methode aus der ersten Feature komplett vernachlässigt werden. Ein Aufruf der umbenannten Methode würde niemals stattfinden.

In Quelltext 2.3 auf der nächsten Seite ist das Ergebnis der Kombination der Klasse *Edge* der beiden Features *Base* und *Weighted* für die Methoden *toString* und *equals*

dargestellt. Beiden Methoden wurden in beiden Features implementiert. Während die Methode *equals* im Feature *Weighted* den Aufruf *original* enthält, ist dies bei der Methode *toString* nicht der Fall. Das kombinierte Feature enthält daher für die Methode *toString* lediglich den Programmcode aus dem Feature *Weighted*. Die Methode *equals* aus dem Feature *Base* wird umbenannt in *equals__wrappee__Base* in das kombinierte Feature aufgenommen. In der Implementierung im Feature *Weighted* wird *original* durch *equals__wrappee__Base* ersetzt und die Methode dann ebenfalls in das kombinierte Feature übernommen.

```
1  public class Edge implements Comparable<Edge> {                Base • Weighted
2       private boolean  equals__wrappee__Base(Object obj){
3           // Aequivalenzberechnung (Feature Base)
4           return true;
5       }
6       public boolean equals(Object obj){
7           // Aequivalenzberechnung (Feature Weighted)
8           equals__wrappee__Base(obj);
9           return true;
10      }
11      public String toString(){
12          // String-Ausgabe (Feature Weighted)
13          return "";
14      }
15 }
```

Quelltext 2.3: Klasse *Edge* des Feature-Moduls zu *Base* • *Weighted* mit gemeinsamen Methoden

Bei Konstruktoren ist dies nicht möglich, da sichergestellt werden muss, dass auch der Programmcode des ersten Features ausgeführt wird. Daher wird in *FeatureHouse* der Code der beiden Konstruktoren hintereinander in den neuen Konstruktor eingefügt. So entsteht der in Quelltext 2.4 dargestellte Programmcode für den Konstruktor.

```
1  public class Edge implements Comparable<Edge> {                Base • Weighted
2       public Edge  (Node node1, Node node2){
3           // Konstruktor Edge(Node,Node) (Feature Base)
4           // Konstruktor Edge(Node,Node) (Feature Weighted)
5       }
6  }
```

Quelltext 2.4: Konstruktor der Klasse *Edge* des Feature-Moduls zu *Base* • *Weighted*

2.2 Spezifikation mithilfe von Design by Contract

Um Software-Produkte zu verifizieren, müssen diese zunächst spezifiziert werden. Eine Möglichkeit der Spezifikation bietet *Design by Contract*. Bei *Design by Contract* werden Kontrakte zwischen der Methode und dem Aufrufer der Methode geschlossen. Diese bestehen aus Vor- und Nachbedingungen [TSK+12]. Vorbedingungen müssen vom Aufrufer der Methode eingehalten werden. Im Gegenzug kann der Aufrufer der Methode davon ausgehen, dass die Nachbedingungen nach der Ausführung der

Methode erfüllt sind, wenn auch die Vorbedingungen erfüllt waren. Weiterhin spezifizieren Klasseninvarianten Eigenschaften, die vor und nach einem Methodenaufruf erfüllt sein müssen [TSK+12].

2.2.1 Vorteile von Design by Contract

Neben einer Möglichkeit der Spezifikation von Software bietet *Design by Contract* weitere Vorteile. Mithilfe von Kontrakten ist eine formale Dokumentation möglich [LC03]. Die Spezifikationen sind abstrakter als Code. Während Programmcode darstellt, wie die Funktionalität erreicht wird, dienen die Spezifikationen dazu, zu beschreiben, welche Funktionalität durch die Methode umgesetzt wird [LC03]. Weiterhin ist es wahrscheinlicher, dass die Spezifikationen mit Änderungen am Programm angepasst werden, als bei informellen Dokumentationen [LC03].

Mit *Blame Assignment* nennen Leavens und Cheon einen weiteren Vorteil von *Desgin by Contract* [LC03]. Kontrakte machen es einfacher, die Ursache eines Fehlers zu erkennen. Wenn die Vorbedingungen nicht erfüllt sind, liegt der Fehler beim Aufrufer. Sind hingegen die Vorbedingungen erfüllt, die Nachbedingungen aber nicht, ist der Fehler innerhalb der Methode.

Weiterhin können Programme effizienter arbeiten, da die Eingabeparameter nicht durch den Programm-Code zur Laufzeit geprüft werden müssen [LC03]. Die entsprechenden Kontrakte werden lediglich zu Verifikation genutzt und im fertigen Software-Produkt nicht beachtet. Außerdem sind diese Überprüfungen nicht immer effizient möglich [LC03]. Um zum Beispiel zu überprüfen, ob eine Liste sortiert ist, muss die komplette Liste durchlaufen und jedes Element mit dem Nachfolger verglichen werden.

Außerdem erleichtern Kontrakte das Verständnis einer Methode [LC03]. Um die Funktionalität einer Methode ohne Kontrakte zu ermitteln, muss der Programmcode analysiert werden. Für die aufgerufenen Methode muss wiederum untersucht werden, welche Funktionalität diese Methoden haben. Wenn jedoch Kontrakte vorhanden sind, müssen nur die Kontrakte der untersuchten Methode analysiert werden [LC03].

2.2.2 Java Modelling Language

Die Java Modeling Language (JML) ist eine Möglichkeit, Javaprogramme mit *Design by Contract* zu spezifizieren. Quelltext 2.5 auf der nächsten Seite zeigt ein Programm mit JML-Spezifikationen. Es handelt sich um einen Ausschnitt der Klasse *Edge* für die Repräsentation einer Kante in einem Graphen. Jede Kante hat ein Gewicht *weight*. Dieses Gewicht muss immer einen Wert größer 0 haben. Mithilfe der Methode *setWeight* kann diesem Gewicht ein Wert zugeordnet werden. Mit der Methode *addWeight* wird das Gewicht um einen angegebenen Wert erhöht und *getWeight* liefert das aktuelle Gewicht der Kante.

Diese Spezifikationen können mithilfe von JML angegeben werden. Sie erfolgen innerhalb von Java-Kommentaren und werden mit @ eingeleitet. Jenachdem, ob es sich bei einer Spezifikation um eine Vorbedingung, Nachbedingung oder Invariante handelt, folgt eines der Stichworte `requires`, `ensures` oder `invariant` und ein, zu Spezifikation gehörender logischer Ausdruck. Abgeschlossen werden die einzelnen

Spezifikationen mit einem Semikolon. Innerhalb dieser Spezifikationen werden aussagenlogische Ausdrücke verwendet. `!` entspricht der Negierung, `==>` der Implikation, `==` der Äquivalenz und `!=` entspricht der Negierung der Äquivalenz.

```
1  public class Edge {
2      //@ invariant weight > 0;
3      private int weight;
4
5      /*@
6        @ requires wgt > 0;
7        @ ensures this.weight == wgt;
8        @*/
9      public void setWeight(int wgt){
10         this.weight = wgt;
11     }
12
13     /*@
14       @ ensures \result == this.weight;
15       @*/
16     public /*@pure@*/ int getWeight(){
17         return this.weight;
18     }
19
20     /*@
21       @ requires wgt >= 0;
22       @ ensures this.weight == \old(this.weight) + wht
23       @     &&    \result == this.weight;
24       @*/
25     public int addWeight(int wgt){
26         this.weight += wgt;
27         return this.weight;
28     }
29 }
```

Quelltext 2.5: Beispielklasse mit Spezifikationen in JML

Mit der Invariante `//@ invariant weight > 0;` in Zeile 2 wird angegeben, dass das Gewicht zu jedem Zeitpunkt einen Wert haben muss, der größer ist als 0.

Mit *requires* werden die Vorbedingungen angegeben. Beim Aufruf der Methode *setWeight* muss beispielsweise das übergebene Gewicht einen Wert größer als 0 haben. *getWeight* besitzt keine Vorbedingungen. Hier fehlt daher das Schlüsselwort *requires*.

Mit `ensures` wird eine Nachbedingung eingeleitet. Diese muss nach der Ausführung der Methode immer erfüllt sein, wenn auch die Vorbedingungen erfüllt waren. Mit dem Schlüsselwort `\result` kann auf das Ergebnis der Methode zurück gegriffen werden. `\old` dient dazu, die Variableninhalte vor dem Aufruf der Methode zu ermitteln.

Mit der Nachbedingung der Methode *setWeight* wird dem Aufrufer zugesichert, dass das Gewicht den übergebenen Wert annimmt. Voraussetzung hierfür ist, dass die Vorbedingung erfüllt war, der übergebene Wert also größer war als 0. Die Nachbedingung der Methode *getWeight* sichert zu, dass das Ergebnis der Methode dem

aktuellen Gewicht entspricht. Bei der Methode *addWeight* wird zum einen zugesichert, dass der übergebene Wert auf das Kantengewicht vor Aufruf der Methode addiert wird, zum anderen, dass das Ergebnis der Methode das neue Kantengewicht ist. Auch hier muss wieder die Vorbedingung erfüllt sein, damit der Aufrufer sich auf diese Zusicherungen verlassen kann.

Ein weiteres Schlüsselwort ist `pure`. Mit diesem Schlüsselwort wird festgelegt, dass keine Änderungen in einzelnen Werten innerhalb des Programms vorgenommen werden. Dadurch ist es möglich, als `pure` gekennzeichnete Methoden in Invarianten, Vor- und Nachbedingungen zu verwenden. Würden durch die Überprüfung von Kontrakten Programm-Werte verändert werden, könnte dies Auswirkungen auf das Programm haben, die nicht gewünscht sind. Würde beispielsweise die Methode *addWeight* in einem Kontrakt verwendet werden, würde bei jeder Überprüfung dieses Kontraktes die Methode aufgerufen und dementsprechend das Gewicht der Kante verändert werden.

2.3 Verifikation mithilfe von Kontrakten

Um ein Software-Produkt mit JML-Spezifikationen zu verifizieren, können verschiedene Anaylse-Verfahren verwendet werden. Zu Ihnen gehören *Model Checking* und *Theorem Proving*.

2.3.1 Model Checking

Model Checking ist eine Technik, um zu verifizieren, dass ein formales Modell eines Systems dessen Spezifikationen entspricht [CGP99]. Während zunächst hauptsächlich abstrakte Modelle von Systemen und Hardware betrachtet wurden, wird Model Checking heute auch zunehmend auf Software Systeme angewendet [TAK+14]. Als Modell wird oft das kompilierte Programm oder eine Abstraktion davon verwendet [CGP99]. Die Spezifikationen definieren Eigenschaften, die das Modell erfüllen muss.

Das Modell besteht aus Zuständen und Transitionen [DKW08]. Zustände werden durch die Werte der einzelnen Variablen, Konfigurationen und Speicherinhalte beschrieben. Transitionen beschreiben den Übergang von einem Zustand in einen anderen. Dies können beispielsweise die einzelnen Methoden sein. Model Checker ermitteln alle erreichbaren Zustände des Modells und prüfen dabei, ob die Spezifikationen erfüllt sind. Entspricht ein Zustand nicht den Spezifikationen, kann ein Gegenbeispiel ermittelt werden, dass angibt, wie der Fehler entstanden ist [DKW08]. Bei einem Programm, das die Klasse *Edge* aus Quelltext 2.5 auf der vorherigen Seite verwendet, würde jeder Zustand beispielsweise die Gewichte aller Kanten enthalten. Transitionen sind dann zum Beispiel die Methoden *setWeight* und *addWeight*.

Model Checking hat jedoch den Nachteil, dass die Anzahl der Zustände sehr schnell steigt. Dies wird auch als Zustandsraum-Explosion bezeichnet [CGP99]. Die Anzahl der erreichbaren Zustände ist bei vielen Software-Systemen bereits nicht mehr managebar. Als Gegenmaßnahme können daher Zustände mit gleichen Eigenschaften verschmolzen werden, sodass diese nicht mehrfach bearbeitet werden müssen. Allerdings erfordert dies, dass alle bisherigen Zustände abgespeichert werden, was wiederum zu Speicherengpässen führen kann.

2.3.2 Theorem Proving

Theorem Proving ist ein deduktiver Ansatz, um die Gültigkeit logischer Formeln zu beweisen [Ehr02]. Für den Beweis der Korrektheit werden Schlussfolgerungen und Transformationen angewendet. Ein *Theorem Prover* ist ein Tool, das diese verwendet, um die logischen Formeln zu beweisen.

Theorem Proving kann auf Programme und deren Spezifikationen angewendet werden. Die Korrektheit einer Methode kann bewiesen werden, indem nachgewiesen wird, dass alle Nachbedingungen erfüllt sind, wenn auch alle Vorbedingungen erfüllt sind.

Wenn dieser Beweis erbracht werden kann, entspricht die Methode den Spezifikationen. Allerdings ist es nicht einfach, die Spezifikationen zu formulieren und Programme mit diesen Spezifikationen zu beweisen. Sowohl für die Spezifikation, als auch für den Beweis werden hoch ausgebildete, spezialisierte Experten mit viel Erfahrung benötigt [CGP99]. Das ein Beweis nicht geschlossen wird, bedeutet außerdem nicht, dass das Programm fehlerhaft sein muss. Daher ist *Theorem Proving* allein nicht ausreichend für die Verifikation von Programmen.

Der große Nachteil von *Theorem Proving* ist, dass es sehr zeitaufwendig ist und Expertenwissen erfordert. In einigen Fällen ist es günstiger, statt dessen beispielsweise Tests durchzuführen, um ein Programm zu verifizieren [Ehr02].

2.4 Spezifikation von Software-Produktlinien

Die Spezifikation von Software-Produktlinien unterscheidet sich von der Spezifikation spezieller Software-Produkte. Statt genau eines Software-Produktes müssen bei Software-Produktlinien mehrere Produkte beschrieben werden.

2.4.1 Arten von Spezifikationen

Die Spezifikationen können unterschiedliche Mengen von Software-Produkten umfassen. Thüm et al. [TAK+14] beschreiben folgende Arten der Spezifikation:

Domänenunabhängige Spezifikation

Domänenunabhängige Spezifikationen werden unabhängig von einer Software-Produktlinie erstellt. Sie werden meist speziell für einzelne Programmiersprachen definiert. Alle in der Programmiersprache geschriebene Programme müssen diese Spezifikationen erfüllen.

Globale Spezifikation

Bei der globalen Spezifikation werden die Eigenschaften spezifiziert, die für alle Software-Produkte gelten. Eigenschaften für spezielle Software-Produkte werden bei dieser Art der Spezifikation nicht angegeben.

Produkt-basierte Spezifikation

Bei der Produkt-basierten Spezifikation werden alle Software-Produkte individuell spezifiziert. Daher ist diese Spezifikationsmethode nur für kleine Software-Produktlinien mit wenigen Software-Produkten sinnvoll. Alternativ können auch nur die Software-Produkte spezifiziert werden, die auch tatsächlich verwendet werden. Produktbasierte Spezifikation sollte nur angewendet werden, wenn die einzelnen Software-Produkte nur wenige Gemeinsamkeiten besitzen.

Feature-basierte Spezifikation

Bei der Feature-basierten Spezifikation werden die einzelnen Features isoliert spezifiziert. Das heißt, sämtliche Beziehungen zu anderen Features werden nicht beachtet. Dadurch können diese Spezifikationen genutzt werden, um Eigenschaften über mehrere Features zu verifizieren und zum Beispiel zu erkennen, wenn Features zwar in Isolation arbeiten, wie beschrieben, das Verhalten aber unerwartet ist, sobald Features kombiniert werden.

Familien-basierte Spezifikation

Bei der Familien-basierten Spezifikation wird die komplette Software-Produktlinie spezifiziert. Dazu müssen sämtliche Features bei der Spezifikation bekannt sein. Die einzelnen Spezifikationen können sich dabei auf bestimmte Feature-Kombinationen beziehen. Sowohl Feature-, als auch Produkt-basierte Spezifikationen sind Teilmenge der Familien-basierten Spezifikationen.

2.4.2 Design By Contract in Feature-Orientierter Programmierung

Bei der Feature-Orientierten Programmierung wird Software in Feature-Module unterteilt. Um ein spezielles Software-Produkt zu erhalten, werden die benötigten Feature-Module kombiniert. Werden die in den Feature-Modulen angegebenen Spezifikationen auch im Software-Produkt benötigt, beispielsweise für die Verifikation des Produktes, müssen diese ebenfalls kombiniert werden. Thüm et al. [TSK+12] beschreiben verschiedene Ansätze für die Komposition von Spezifikationen in Feature-Modulen. Weitere Kompositionsverfahren werden von Benduhn [Ben12] beschrieben. Thüm et al. [TBA+13] geben einen Überblick über einige der Verfahren.

Plain Contracting

Beim *Plain Contracting* erfolgt die Spezifikationen nur in dem Feature, in dem die zu spezifizierende Methode eingeführt wird [TSK+12, Ben12, TBA+13]. Wird diese Methode durch ein anderes Feature neu implementiert, muss diese Implementierung die gleichen Spezifikationen erfüllen. Jede Methode wird dadurch nur einmal spezifiziert. Dies erleichtert die Entwicklung, da der Entwickler nur genau eine Spezifikation zu einer verwendeten Methode kennen muss. Allerdings muss er suchen, in welchem Feature die Spezifikation erstellt wurde. Weiterhin schränkt dies die Entwicklung auch ein, da manche Redefinitionen einer Methode die Anpassung der Kontrakte erfordern oder Nachbedingungen erlauben, die Vorteile bei der Verwendung der Methode haben.

Cumulative Contract Refinement

Werden Kontrakte einer Methode mittels *Cumulative Contract Refinement* kombiniert, werden die Vorbedingungen mit `ODER` verknüpft, die Nachbedingungen mit `UND` [Ben12, TBA+13]. Es ist also ausreichend, wenn der Aufrufer dafür sorgt, dass die Vorbedingungen aus einem der kombinierten Features erfüllt werden. Im Gegensatz dazu muss die Methode immer alle Nachbedingungen erfüllen, der Aufrufer erhält also viele Garantien.

Conjunctive Contract Refinement

Beim *Conjunctive Contract Refinement* werden alle Vorbedingungen mit `UND` verknüpft. Ebenso wird mit den Nachbedingungen der Methode verfahren [TSK+12, Ben12, TBA+13]. Wie beim *Cumulative Contracting* werden so viele Garantien gegeben. Allerdings muss der Aufrufer auch dafür sorgen, dass alle Vorbedingungen erfüllt sind.

Consecutive Contract Refinement

Beim *Consecutive Contract Refinement* können weitere Features, die eine Methode redefinieren, die dazugehörigen Spezifikationen nur erweitern. Die redefinierte Methode muss dann sowohl die alte Spezifikation als auch die neue Spezifikation erfüllen [Ben12, TBA+13]. Da die Kontrakte durch andere Features nicht entfernt werden können, kann sich bei der Verwendung der Elemente darauf verlassen werden, dass die im gleichen Feature spezifizierten Eigenschaften erfüllt werden. Allerdings können die Spezifikationen nicht aufgeweicht werden. Dies könnte jedoch nötig sein.

Explicit Contract Refinement

Beim *Explicit Contract Refinement* werden Spezifikationen einer Methode durch die Spezifikationen der Methode im neuen Feature ersetzt [TSK+12, Ben12, TBA+13]. Allerdings kann über ein Schlüsselwort auf die ursprüngliche Spezifikation zurückgegriffen werden. Die Verwendung der ursprünglichen Spezifikation ist jedoch nicht verpflichtend. Es ist auch möglich, die alte Spezifikation nur in den Vor- oder Nachbedingungen zu referenzieren. Der Nachteil des *Explicit Contract Refinement* liegt jedoch in der Komplexität. Wenn mehrere Features eine Spezifikation verfeinern, einige sie jedoch überschreiben, kann es schwierig sein, zu erkennen, welche Bedingungen erfüllt sein müssen und welche nicht.

Method Based Contract Refinement

Häufig wird genau eines dieser Verfahren in einer Software-Produktlinie angewendet. Weigelt [Wei13] erklärt jedoch, dass es von Vorteil wäre, je Methode anzugeben nach welchem Kompositionsverfahren gearbeitet werden soll. Um dies bei Kontrakten in JML zu ermöglichen, hat er Schlüsselwörter vorgeschlagen, welche das Verfahren angeben. Ist kein Schlüsselwort vorhanden, wird *Explicit Contract Refinement* angewendet. Für *Cumulative*, *Conjunctive* und *Consecutive Contracting* führt er die Schlüsselwörter *`cumulative_contract`*, *`conjunctive_contract`* und *`consecutive_contract`* ein. Für *Plain Contracting* schlägt er zwei Schlüsselwörter vor:

final_contract und *final_method*. Das Schlüsselwort *final_method* verbietet zusätzlich zum Überschreiben der Kontrakte auch das Überschreiben der Methode.

Weiterhin hat Weigelt Regeln beschrieben, nach denen das angewendete Verfahren während der Komposition geändert werden kann [Wei13]. Abbildung 2.5 stellt dar, welche Wechsel erlaubt sind. Das Verändern ist dabei nur in eine Richtung möglich. Da *Explicit Contracting* das allgemeinste Verfahren ist, ist ein Wechsel zu einem der anderen Vorgehen möglich. Beim Wechsel des Kompostionsverfahrens können auch einzelne Verfahren übersprungen werden. Wurde jedoch einmal mit *Plain Contracting* eine Spezifikation angegeben, kann das Kompositionsverfahren nicht mehr geändert werden.

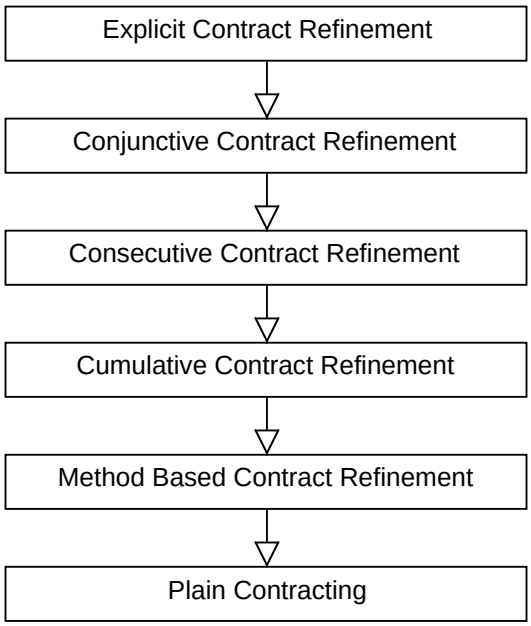

Abbildung 2.5: Rangfolge der Kompositionsverfahren bei *Method Based Contract Refinement*

2.5 Verifikation von Software-Produktlinien

Software-Produktlinien beschreiben eine Menge von Software-Produkten. Um dies zu gewährleisten, muss bei der Verifikation von Software-Produktlinien sichergestellt werden, dass alle Software-Produkte verifiziert werden. Um sicherzustellen, dass alle erzeugen Software-Produkte zur Laufzeit korrekt arbeiten, existieren verschiedene Herangehensweisen. Zu diesen gehören Produkt-basierte, Familien-basierte und Feature-basierte Analysen [TAK+14].

Produkt-basiert

Bei der Produkt-basierten Analyse werden einzelne Software-Produkte generiert und verifiziert[TAK+14]. Dadurch können bereits vorhandene Verifikations-Methoden und -Tools genutzt werden. Allerdings erfordert die Verifikation aller möglichen

Software-Produkte auch die Generierung dieser. Viele Produkte unterscheiden sich dabei aber nur durch wenige Features, sodass bereits bei Software-Produktlinien mit wenigen Features viele Verifikationsschritte mehrfach durchlaufen werden. Weiterhin steigt die Anzahl der möglichen Produkte bis zu exponentiell in der Anzahl der Features an, sodass bereits bei 10 Features $2^{10} = 1024$ mögliche Feature-Kombinationen existieren. Selbst wenn nicht alle dieser Feature-Kombinationen möglich sind, bleibt die Anzahl der zu verifizierenden Software-Produkte sehr hoch.

Familien-basiert

Familien-basierte Analyse arbeitet mit Produktmengen und gültigen Kombinationen davon [TAK+14]. Im Gegensatz zur Produkt-basierten Analyse muss nicht jedes Produkt generiert werden, da eine Menge von Produkten betrachtet wird. So werden redundante Berechnungen aufgrund von Gemeinsamkeiten einzelner Software-Produkte vermieden [TAK+14]. Allerdings können bereits vorhandene Tools für einzelne Software-Produkte nicht ohne weiteres angewendet werden [TAK+14]. Weiterhin muss bei der Änderung eines Features die komplette Software-Produktlinie erneut analysiert werden [TAK+14]. Auch die Änderung des Feature-Modells der Software-Produktlinie erfordert meist eine erneute Analyse der kompletten Software-Produktlinie. Da alle relevanten Produkte zusammen analysiert werden, kann die Analyse leicht die Speichergrenzen erreichen [TAK+14].

Eine Möglichkeit die Familien-basierte Verifikation einer Software-Produktlinie zu ermöglichen, bietet die Generierung eines Metaproduktes. Anstatt alle Produkte ausgehend von einer Konfiguration zu generieren, wird ein Programm erzeugt, dass alle Software-Produkte simulieren kann. Um dies zu ermöglichen, erfolgt die Festlegung der Konfiguration erst zur Laufzeit. Post und Sinz bezeichnen dieses Vorgehen als *Configuration Lifting* [PS08]. Um eine Software-Produktlinie zu verifizieren, kann das Metaprodukt dieser Software-Produktlinie genutzt werden.

Feature-basiert

Bei der Feature-basierten Analyse werden einzelne Features in Isolation analysiert [TAK+14]. Da mit einzelnen Features gearbeitet wird, ist es nicht nötig, alle möglichen Produkte zu generieren. Dadurch sind auch keine redundanten Berechnungen nötig [TAK+14]. Da alle Features isoliert betrachtet werden, ist kein Wissen über andere Features und deren Beziehungen nötig [TAK+14]. Wenn einzelnen Features sich ändern, müssen nur die geänderten Features erneut analysiert werden. Da die Features wesentlich kleiner sind als komplette Software-Produkte oder Software-Produktlinien, können auch aufwendigere Analyseverfahren verwendet werden, die bei Familien- oder Produkt-basierten Analysen aufgrund des Ressourcenbedarfs nicht anwendbar wären. Allerdings hat die isolierte Betrachtung der Features auch den Nachteil, dass Probleme zwischen den Features nicht erkannt werden. Ein bekanntes Problem sind Feature-Beeinflussungen [CKMRM03]. Hierbei arbeiten Features in Isolation so wie erwartet, zeigen aber ein unerwartetes Verhalten bei deren Kombination.

3. Kontrakt-Komposition und Optimierung im Metaprodukt

Um die Familien-basierte Verifikation einer Software-Produktlinie zu ermöglichen, kann ein Metaprodukt erzeugt werden. Dieses Metaprodukt simuliert alle Software-Produkte, die mit der Software-Produktlinie erzeugt werden können. Daher kann die Familien-basierte Verifikation mithilfe des Metaproduktes erfolgen. Um bei der Verifikation auch die Spezifikationen nutzen zu können, ist es sinnvoll auch diese in das Metaprodukt zu überführen, sodass die zur Laufzeit gewählte Konfiguration beachtet wird.

Meinicke hat in seiner Arbeit bereits ein Metaprodukt erzeugt [Mei13]. Er hat sich dabei auf Software-Produktlinien beschränkt, die in Java geschrieben wurden und Spezifikationen in Form von JML-Anotationen enthalten. Weiterhin hat er sich auf das Kompositionsverfahren *Explicit Contract Refinement* konzentriert. Dieses Metaprodukt enthält Redundanzen in den Spezifikationen. Außerdem lassen sich die Spezifikationen vereinfachen, wenn Beziehungen zwischen Features beachtet werden. Aufbauend auf der Arbeit von Meinicke wird die Metaprodukt-Erzeugung um weitere Kompositionsverfahren sowie Optimierungen in den erzeugten Spezifikationen erweitert. Die Optimierungen haben zum einen Einfluss auf die Lesbarkeit der erzeugten Spezifikationen, zum anderen könnten sie die Effizienz der eingesetzten Verifikationstools beeinflussen.

In Abschnitt 3.1 wird zunächst beschrieben, wie Meinicke die Variabilität von der Compilezeit in die Laufzeit überführt hat. Das Konzept von Meinicke zur Komposition von Kontrakten im Metaprodukt mithilfe von *Explicit Contract Refinement* wird in Abschnitt 3.2.1 beschrieben. In Abschnitt 3.2.2 folgen jeweils Optimierungspotentiale für die von Meinicke beschriebenen Spezifikationen im Metaprodukt. In den Kapiteln Abschnitt 3.3 bis Abschnitt 3.7 werden Verfahren für Komposition von Kontrakten für die Metaprodukt-Generierung mittels *Conjunctive Contract Refinement*, *Consecutive Contract Refinement*, *Cumulative Contract Refinement*, *Plain Contracting* und *Method Based Contract Refinement* vorgeschlagen. Dabei wird zunächst allgemein beschrieben, wie bei der Kontrakt-Komposition für die Metapro-

dukt-Generierung vorgegangen werden kann. Anschließend werden die so erzeugten Spezifikationen betrachtet und Optimierungspotentiale dargestellt.

3.1 Kodierung der Variabilität zur Laufzeit

Das Metaprodukt muss zur Laufzeit das durch die festgelegte Konfiguration definierte Software-Produkt simulieren. Um dies zu ermöglichen, muss die Konfiguration zur Laufzeit definiert und abrufbar sein. Weiterhin muss beim Aufruf einer Methode der zu der gewählten Konfiguration gehörenden Programmcode ausgeführt werden. Weiterhin muss sichergestellt werden, dass die gewählte Konfiguration gültig ist und die aufgerufene Methode in dem simulierten Software-Produkt vorhanden ist.

Dieses Kapitel bezieht sich auf die von Meinicke [Mei13] beschriebene Erzeugung des Metaproduktes. Thüm et al. [TSHA12] beschreiben ein analoges Vorgehen.

3.1.1 FeatureModel-Klasse

Um die Konfiguration zur Laufzeit prüfen und nutzen zu können, wird eine zusätzliche Klasse bei der Generierung des Metaproduktes erzeugt. Ein Software-Produkt wird durch seine Konfiguration beschrieben. Um alle Produkte abbilden zu können, muss diese Klasse daher zwei Aufgaben erfüllen:

1. Pro Feature muss ein Feld vorhanden sein, dass den Selektionsstatus des Features abbildet.
2. Eine Methode muss die Gültigkeit der so abgebildeten Konfiguration ermitteln.

Der Selektionsstatus wird durch Wahrheitswerte abgebildet. Ist ein Feature in der Konfiguration gewählt, ist der Wert des entsprechenden Feldes *wahr*, ansonsten *falsch*. Meinicke schlägt vor, den Wert zunächst undefiniert zu lassen, sodass Verifikationstools den Feldern beide möglichen Werte zuweisen [Mei13]. Ausgenommen hiervon sind Kern-Features und tote Features. Kern-Features sind in jedem Software-Produkt enthalten. Daher kann der Wahrheitswert des entsprechenden Feldes immer mit *wahr* initialisiert werden. Tote Features sind Features, die in keiner gültigen Konfiguration und somit in keinem Software-Produkt vorhanden sind. Diese können mit *falsch* initialisiert werden. Tote Features sollten in einer Software-Produktlinie nicht vorhanden sein.

Für die Ermittlung der Gültigkeit wird die aussagenlogische Repräsentation des Feature-Modells verwendet. Um diese Methode in Spezifikationen verwenden zu können, wird die Methode mit `pure` gekennzeichnet.

Alle Felder und die Methode müssen statisch sein, sodass kein Objekt der Klasse erzeugt und verwaltet werden muss. Die so erzeugte Klasse kann verwendet werden, um eine Konfiguration festzulegen und so das entsprechende Software-Produkt zu simulieren. In Quelltext 3.1 auf der nächsten Seite ist die Klasse *FeatureModel* dargestellt. Diese kodiert die Variabilität für die in Abschnitt 2.1.1 beschriebene Software-Produktlinie. Für alle Features wurde eine Bool'sche Variable angelegt. Den Variablen zu den Features *GPLscratchFHJML*, *Base* und *Direction* wurden die

Wahrheitswerte `true` zugeordnet, da diese Features Kern-Features sind. Alle anderen Variablen bleiben uninitialisiert. Die Methode *valid* prüft, ob die gewählte Konfiguration valide ist. Dazu wird die aussagenlogische Repräsentation des Feature-Modells verwendet. Da diese in dem Metaprodukt sehr lang ist, wurde sie abgekürzt.

```
1  public class FeatureModel {
2
3      public static boolean uniquenodes;
4      public static boolean optimalconnection;
5      public static boolean gplscratchfhjml = true;
6      public static boolean base = true;
7      public static boolean undirected;
8      public static boolean uniqueedges;
9      public static boolean unique;
10     public static boolean shortestconnection;
11     public static boolean algorithm;
12     public static boolean maxedges;
13     public static boolean direction = true;
14     public static boolean directed;
15     public static boolean weighted;
16     public static boolean connection;
17
18     public /*@pure@*/ static boolean valid() {
19         return gplscratchfhjml  &&  /* ... */;
20     }
21 }
```

Quelltext 3.1: Kodierung der Variabilität einer Software-Produktlinie für Graphen

In den folgenden Abschnitten bezeichnet *FeatureModel* diese Klasse. Die einzelnen Felder erhalten dabei den gleichen Namen, wie die repräsentierten Features. Die Ermittlung der Gültigkeit einer Konfiguration erfolgt mit der Methode *vaild*.

3.1.2 Komposition von Methoden

Beim Aufruf einer Methode im Metaprodukt muss sichergestellt werden, dass der Programmcode, der zu der gewählten Konfiguration gehört, ausgeführt wird. Um dies zu gewährleisten, muss die Komposition von Methoden zunächst genauso erfolgen, wie bei der Erstellung eines Software-Produktes. Das bedeutet, bei der Redefinition einer Methode wird die alte Methode umbenannt, die redefinierende Methode in die Klasse eingefügt und der Aufruf von *original* in der redefinierenden Methode durch den neuen Namen der alten Methode ersetzt.

Dies deckt den Fall ab, dass das neue Feature in der gewählten Konfiguration vorhanden ist. Wenn das Feature nicht vorhanden ist, muss die alte Methode aufgerufen werden. Um dies zu realisieren, kann auf die Klasse *FeatureModel* zurückgegriffen werden. Wenn das Feature nicht gewählt wurde, muss die alte Methode ausgeführt und danach die Methode verlassen werden.

Ein Beispiel soll dies verdeutlichen. In Quelltext 3.2 auf der nächsten Seite sind die Implementierungen der Methode *equals* in den Features *Base* und *Weighted* abgebildet. Wenn die beiden Features komponiert werden entsteht im Metaprodukt

der abgebildete Programmcode. Die in *Base* definierte Methode wird umbenannt in `equals__wrappee__Base`. Anschließend wird die Implementierung aus dem Feature *Weighted* eingefügt. Bis zu diesem Punkt entspricht das Vorgehen dem gleichen, wie bei der Erstellung eines speziellen Software-Produktes. Im Metaprodukt wird zusätzlich der grau hervorgehobene Programmcode eingefügt. Wenn das Feature *Weighted* in der Konfiguration nicht vorhanden ist, muss die Implementierung aus dem Feature *Base* ausgeführt und das Ergebnis ausgegeben werden.

Base

```
1  public  class Edge implements Comparable<Edge> {
2
3      private boolean  equals(Object ob) {
4          return (ob instanceof Edge) ? true : false;
5      }
6  }
```

Weighted

```
9  public  class Edge implements Comparable<Edge> {
10
11     private boolean  equals(Object ob) {
12         if (original(ob) && weight == ((Edge)ob).weight)
13             return true;
14         return false;
15     }
16 }
```

Metaprodukt

```
17 public  class Edge implements Comparable<Edge> {
18
19     private boolean  equals__wrappee__Base(Object ob) {
20         return (ob instanceof Edge) ? true : false;
21     }
22
23     private boolean  equals(Object ob) {
24         if (!FM.FeatureModel.weighted)
25             return equals__wrappee__Base(ob);
26         if (equals__wrappee__Base(ob) && weight == ((Edge)ob).weight)
27             return true;
28         return false;
29     }
30 }
```

Quelltext 3.2: Kodierung der Variabilität im Metaprodukt in Methoden am Beispiel der Methode *equals* der Features *Base* und *Weighted*

3.1.3 Allgemeine Spezifikationen

Bei der Verifikation des Metaproduktes muss beachtet werden, dass nicht alle Konfigurationen valide sind. Außerdem enthält das Metaprodukt alle Methoden, die in den Feature-Modulen definiert wurden. Eine Methode muss aber nicht in allen Konfigurationen vorhanden sein.

Um sicherzustellen, dass nur valide Konfigurationen verifiziert werden, können verschiedene Vorgehen gewählt werden. Zum einen kann ein genutztes Verifikationstool

sicherstellen, dass nur valide Konfigurationen betrachtet werden. Zum anderen kann bei Software-Produktlinien mit Kontrakten eine zusätzliche Vorbedingung erstellt werden, die garantiert, dass die Konfiguration gültig ist. In beiden Fällen kann die Methode *valid* der zusätzlich generierten Klasse *FeatureModel* genutzt werden, um zu prüfen, ob die Konfiguration gültig ist.

Um Methoden nur für Konfigurationen zu verifizieren, die die Methode enthalten, kann eine zusätzliche Vorbedingung erzeugt werden. Bei dieser Vorbedingung handelt es sich um die Disjunktion aller Selektionsstati der Features, die die spezifizierte Methode enthalten. Bei jeder Komposition kann diese Disjunktion um das neue Feature ergänzt werden.

In Quelltext 3.3 sind die so erzeugten zusätzlichen Vorbedingungen abgebildet, die bei der Komposition der Features *Base* und *Weighted* erstellt werden. Sowohl die Methode aus dem Feature *Base*, als auch die komponierte Methode erhalten als zusätzliche Vorbedingung `FeatureModel.valid()` um sicherzustellen, dass die Konfiguration valide ist. Damit die ursprüngliche Methode aufgerufen werden kann, muss das Feature *Base* in der Konfiguration vorhanden sein. Daher wird zusätzlich die Vorbedingung `FeatureModel.base` eingefügt. Die komponierte Methode kann aufgerufen werden, wenn entweder *Base* oder *Weighted* vorhanden ist. Bei dieser Methode wird daher die Vorbedingung `FeatureModel.base || FeatureModel.weighted` eingefügt.

Metaprodukt

```
1  public  class Edge implements Comparable<Edge> {
2
3      /*@
4        @ requires FeatureModel.valid();
5        @ requires FeatureModel.base;
6        @*/
7      private boolean  equals__wrappee__Base(Object ob) {
8          return (ob instanceof Edge) ? true : false;
9      }
10
11     /*@
12       @ requires FeatureModel.valid();
13       @ requires FeatureModel.base || FeatureModel.weighted;
14       @*/
15     private boolean  equals(Object ob) {
16         if (!FM.FeatureModel.weighted)
17             return equals__wrappee__Base(ob);
18         if (equals__wrappee__Base(ob) && weight == ((Edge)ob).weight)
19             return true;
20         return false;
21     }
22 }
```

Quelltext 3.3: Zusätzliche Vorbedingungen im Metaprodukt am Beispiel der Methode *equals* der Features *Base* und *Weighted*

Optimierungspotential

Wenn das Feature-Model bekannt ist, lassen sich diese erzeugten Spezifikationen vereinfachen. Bisher wird die Disjunktion immer erzeugt. Wenn diese Disjunkti-

on jedoch ein Kern-Feature enthält, ist die Vorbedingung immer erfüllt. Da Kern-Features in jeder gültigen Konfiguration vorhanden sind, ist der Selektionsstatus des Features immer wahr. In dem Beispiel geht aus dem Feature-Model hervor, dass das Feature *Base* ein Kern-Feature ist. Wenn das Feature in einer Konfiguration nicht vorhanden ist, würde die Vorbedingung `FeatureModel.valid()` verletzt werden. Aus diesem Grund kann die Disjunktion bei dem Beispiel aus den Vorbedingungen entfernt werden.

In den folgenden Kapiteln werden diese beiden zusätzlichen Vorbedingungen nicht mit in die Spezifikationen aufgenommen, da diese unabhängig vom gewählten Kompositionsverfahren für Kontrakte sind. Bei der Erzeugung eines Metaproduktes müssen diese Klauseln dennoch eingefügt werden, falls dies nötig ist.

3.2 Explicit Contract Refinement

Die Komposition der Spezifikationen mittels *Explicit Contract Refinement* bei der Erzeugung hat Meinicke bereits betrachtet [Mei13]. Thüm et al. [TSHA12] beschreiben die gleichen Regeln für die Komposition. Diese Regeln werden zunächst in Abschnitt 3.2.1 beschrieben. Die so erzeugte Spezifikation kann jedoch optimiert werden. Dieses Optimierungspotential wird in Abschnitt 3.2.2 betrachtet.

3.2.1 Komposition bei der Metaprodukt-Generierung

Meinicke [Mei13] sowie Thüm et al. [TSHA12] beschreiben bereits die Komposition von Kontrakten mittels *Explicit Contract Refinement* für die Erzeugung eines Metaproduktes. Die Spezifikation der Methode im Metaprodukt besteht aus zwei alternativen Spezifikationen. Eine Spezifikation beschreibt den Fall, wenn das verfeinernde Feature gewählt ist, die zweite, wenn es nicht gewählt ist. Wenn das Feature nicht in der Konfiguration vorhanden ist, muss die ursprüngliche Spezifikation gelten. Um Sicherzustellen, dass eine Klausel nur gilt, wenn das Feature vorhanden ist, in dem die Klausel spezifiziert wurde, wird eine Implikationen der Form

```
FeatureModel.<FeatureName> ==>
```

vor den Bedingungen eingefügt. Ist das Feature gewählt nimmt der aussagenlogische Ausdruck `FeatureModel.<FeatureName>` den Wert `wahr` an und der ursprüngliche Kontrakt muss erfüllt sein, damit der neue Kontrakt erfüllt wird. Wenn das Feature nicht ausgewählt ist, hat `FeatureModel.<FeatureName>` den Wert `falsch`. Durch die Implikation ist diese Klausel dann immer erfüllt, unabhängig von der ursprünglichen Bedingung. Nach der Komposition müssen die ursprünglichen Vor- und Nachbedingungen gelten, wenn das Feature nicht gewählt wurde. Daher wird die Implikation

```
!FeatureModel.<FeatureName> ==>
```

vor die ursprünglichen Bedingungen eingefügt. Ist das Feature gewählt, ist die Implikation immer wahr. Wenn das Feature nicht gewählt ist, entspricht der Wahrheitswert dem der ursprünglichen Bedingung. In den neu hinzuzufügenden Bedingungen wird dann das Schlüsselwort ***`original`*** durch die ursprünglichen Bedingungen ersetzt.

In Quelltext 3.4 sind die Spezifikationen in JML für die Methode *addEdge* der Klasse *Graph* in den Features *Base* und *MaxEdges* dargestellt. Das Besipiel stammt aus der, in Abschnitt 2.1.1 beschriebenen Software-Produktlinie für Graphen. Mit der Implementierung in Feature *Base* wird eine Kante in den Graphen eingefügt. Durch die Nachbedingung `hasEdge(edge)` in Zeile 5 wird dies garantiert. Damit dies aber geschehen kann, darf die Kante nicht `null` sein und die beiden verknüpften Knoten müssen vorhanden sein. Dies wird durch die Vorbedingung in den Zeilen 2 bis 4 definiert.

Durch das Feature *MaxEdges* wird eine maximale Kantenzahl eingefügt. Dies beeinflusst auch die Spezifikation der Methode `addEdge`. Zusätzlich zu den bereits vorhandenen Vorbedingungen muss nun auch die maximale Anzahl der Kanten definiert sein. Dies wird in der Spezifikation durch `maxEdges != null` realisiert. Weiterhin wird die Kante nur eingefügt, wenn die maximale Kantenanzahl noch nicht erreicht wurde. In der Spezifikation wird dies in der Nachbedingung beschrieben.

Base

```
1  /*@
2    @ requires edge != null
3    @     && nodes.contains(edge.first)
4    @     && nodes.contains(edge.second);
5    @ ensures hasEdge(edge);
6    @*/
7  public void addEdge(Edge edge) { ... }
```

MaxEdges

```
8  /*@
9    @ requires \original
10   @     && MAXEDGES != null;
11   @ ensures \old(countEdges()) < MAXEDGES ==> \original;
12   @*/
13 public void addEdge(Edge edge) { ... }
```

Quelltext 3.4: Spezifikation der Methode *addEdge* der Klasse *Graph* in den Features *Base* und *MaxEdges*

Bei der Komposition mit den beschriebenen Regeln für *Explicit Contract Refinement* entstehen die in Quelltext 3.5 auf der nächsten Seite dargestellten Spezifikationen im Metaprodukt. Die Vorbedingung in den Zeilen 2 bis 6 beschreibt den Fall, dass das Feature *Base* vorhanden ist, das Feature *MaxEdges* aber nicht. In diesem Fall gelten die in Quelltext 3.4 (Zeile 2 bis 4) dargestellten Vorbedingungen. Da die Methode `addEdge` durch das Feature *Base* eingeführt wird, sind keine Bedingungen für den Fall, dass das Feature *Base* nicht gewählt ist, vorhanden. Die zweite Vorbedingung in den Zeilen 7 bis 12 in Quelltext 3.5 auf der nächsten Seite beschreibt die Spezifikation für den Fall, dass das Feature *MaxEdges* vorhanden ist. Hier gilt die Vorbedingung des Features *MaxEdges*, wobei jedes Auftreten von ***original*** durch die Vorbedingungen ersetzt werden, die vor der Komposition vorhanden waren. Dies wurde dunkelgrau hervorgehoben.

Ähnlich verhält es sich mit den Nachbedingungen. Die Nachbedingung in den Zeile 13 und 14 bildet den Fall ab, dass nur das Feature *Base* gewählt wurde. In der zweiten Nachbedingung (Zeilen 15 bis 17) ist das Feature *MaxEdges* vorhanden.

In diesem Fall gelten die Nachbedingungen, die in Feature *MaxEdges* spezifiziert wurden, wobei wiederum ***original*** ersetzt wird.

Metaprodukt

```
1  /*@
2    @ requires !FeatureModel.maxedges ==>
3    @          FeatureModel.base ==>
4    @              (       edge != null
5    @              &&      nodes.contains(edge.first)
6    @              &&      nodes.contains(edge.second));
7    @ requires FeatureModel.maxedges ==> ((
8    @          FM.FeatureModel.base ==>
9    @              (       edge != null
10   @              &&      nodes.contains(edge.first)
11   @              &&      nodes.contains(edge.second))
12   @              ) &&    MAXEDGES != null);
13   @ ensures !FeatureModel.maxedges ==>
14   @          FeatureModel.base ==> (hasEdge(edge));
15   @ ensures FeatureModel.maxedges ==>
16   @          (\old(countEdges()) < MAXEDGES ==>
17   @                  ( FeatureModel.base ==> (hasEdge(edge)) ));
18   @*/
19 public void addEdge(Edge edge) { /* ... */ }
```

Quelltext 3.5: Spezifikation der Methode *addEdge* im Metaprodukt nach der Komposition von *Base* und *MaxEdges* mittels *Explicit Contract Refinement*

Wie Thüm et al. [TSHA12] und Meinicke [Mei13] gezeigt haben, ist es mit dem so erzeugten Metaprodukt möglich, Software-Produktlinien Familien-basiert zu verifizieren. Durch die beiden in Quelltext 3.5 dunkelgrau hervorgehobenen Bereiche wird jedoch deutlich, dass die Spezifikationen auch Redundanzen enthalten können.

3.2.2 Optimierungspotential

Die nach dem, in Abschnitt 3.2.1 beschriebenen Verfahren erzeugte Spezifikation im Metaprodukt lässt sich noch vereinfachen. Einige Features sind immer vorhanden, wenn die Methode aufgerufen wird. Die Spezifikation in Quelltext 3.5 kann so verändert werden, dass die dunkelgrau hervorgehobene Wiederholung in den Vorbedingungen nur noch einmal vorkommt. Weiterhin entstehen Klauseln für Feature-Kombinationen, die nicht möglich sind. Diese Klauseln sind immer wahr und können daher entfernt werden. Außerdem werden einige Selektionsstati von anderen impliziert, sodass einige Klauseln gekürzt werden können.

Beachtung von Pflicht-Features

Einige Features sind jedem Software-Produkt, in dem eine bestimmte Methode existiert, enthalten. Im Folgenden werden diese Features als **Methoden-Kern-Features** bezeichnet. Wenn diese Features bekannt sind, kann die Spezifikation im Metaprodukt vereinfacht werden. Die Klauseln, die gelten, wenn eines dieser Features nicht vorhanden ist, müssen nicht in die Spezifikation aufgenommen werden. Der Selektionsstatus der Features wäre beim Aufruf der Methode immer `wahr`. Sollte der Selektionsstatus nicht `wahr` sein, weil beispielsweise eine Konfiguration gewählt

ist, die die Methode nicht enthält, ist bereits eine der in Abschnitt 3.1.3 beschriebenen Vorbedingungen verletzt. Ist das Feature aber in der Konfiguration enthalten, hat die erzeugte Klausel die Form `falsch ==> <Bedingung>`. Diese Klausel ist immer erfüllt, da gilt $(falsch \Rightarrow a) \equiv wahr$. Daher muss die Klausel nicht in die Spezifikation aufgenommen werden.

Auch die Klauseln für den Fall, dass das Feature gewählt ist, können vereinfacht werden. Wenn ein Feature beim Aufruf einer Methode immer vorhanden ist, haben die erzeugten Klauseln die Form `wahr ==> <Bedingung>`. Da $(wahr \Rightarrow a) \equiv a$, kann in diesem Fall die Implikation entfernt werden.

Reduzierung von Redundanzen durch das Schlüsselwort original

Die in Quelltext 3.5 auf der vorherigen Seite dunkelgrau hervorgehobene Wiederholung resultiert aus der Verwendung von ***original*** im Feature *MaxEdges*. In dem Feature ist ***original*** durch eine Konjunktion mit den anderen Vorbedingungen verknüpft. Daher muss die ursprüngliche Spezifikation sowohl gelten, wenn das Feature *MaxEdges* gewählt ist, als auch, wenn dies nicht der Fall ist.

Die ursprüngliche Spezifikation kann daher in die komponierte Spezifikation übernommen werden, ohne die Implikation `!FeatureModel.<FeatureName> ==>` einzufügen. Die ursprünglichen Spezifikationen müssen dann sowohl gelten, wenn das Feature *<FeatureName>* gewählt ist, als auch wenn es nicht vorhanden ist. Sind diese Spezifikationen erfüllt, würde auch jedes Auftreten von ***original*** immer den Wahrheitswert `wahr` haben. Daher müssen nur noch die neuen Bedingungen erfüllt werden.

Zu Beachten ist jedoch, dass Vor- und Nachbedingungen separat betrachtet werden müssen. Eine Konjunktion von ***original*** mit den Vorbedingungen bedeutet nicht, dass dies auch bei den Nachbedingungen der Fall ist.

Eliminierung von Tautologien

Durch die Komposition mehrerer Features können umfangreiche Implikationen entstehen. Das Vorhandensein bestimmter Featurekombinationen impliziert bestimmte Bedingungen. Da diese Implikationen aber generiert werden, ist es möglich, dass einige Kombinationen aufgrund des Feature-Modells nicht erlaubt sind. In diesem Fall ist die resultierende Bedingung immer wahr. In einer Software-Produktlinie mit den Features A und B könnte beispielsweise folgende Vorbedingung generiert werden:

```
requires !FeatureModel.b ==> (FeatureModel.a ==> VorbedingungA);
```

Wenn aber aus dem Feature-Modell hervorgeht, dass das Feature A nicht gewählt werden kann, wenn das Feature B nicht gewählt ist, kann diese Bedingung niemals falsch werden. Ist das Feature B vorhanden, entsteht die Bedingung:

```
requires false ==> (FeatureModel.a ==> VorbedingungA);
```

Diese ist immer wahr, unabhängig, ob das Feature A vorhanden ist oder nicht, da $(false \Rightarrow x) \equiv true$ gilt. Ist das Feature B nicht vorhanden, ist aufgrund des Feature-Modells auch Feature A nicht vorhanden, sodass folgende Bedingung entsteht:

```
requires true ==> (false ==> VorbedingungA);
```

Auch diese Bedingung ist immer wahr. Eine ungültige Konfiguration würde die generierte Vorbedingung

```
requires FeatureModel.valid();
```

verletzen, sodass nicht alle Vorbedingungen erfüllt wären.

Da eine so generierte Vorbedingung immer erfüllt ist, wenn eine gültige Konfiguration vorhanden ist und ungültige Konfigurationen bereits eine andere Vorbedingung verletzen, können diese Bedingungen aus dem Metaprodukt entfernt werden.

Kürzung von Implikationen

Weiterhin lassen sich die Implikationen auch kürzen. Wenn bei dem Beispiel mit den Features A und B aus dem Feature-Modell hervorgeht, dass das A immer selektiert ist, wenn B nicht vorhanden ist, könnte aus der generierten Vorbedingung

```
requires !FeatureModel.b ==> FeatureModel.a ==> VorbedingungA;
```

`FeatureModel.a ==>` entfernt werden, da `FeatureModel.a` wahr wäre und $(true \Rightarrow x) \equiv x$ gilt. Die Bedingung kann dann reduziert werden zu:

```
requires !FeatureModel.b ==> VorbedingungA;
```

Die Reihenfolge der Selektionsstati in der Reihe der Implikationen spielt dabei keine Rolle. Die Bedingung am Ende der Implikationskette muss erfüllt sein, wenn alle Features den richtigen Selektionsstatus haben. Es lässt sich leicht zeigen, dass $(a \Rightarrow (b \Rightarrow c)) \equiv (b \Rightarrow (a \Rightarrow c))$. Zwei, in der Reihe aufeinander folgende Selektionsstati können daher vertauscht werden. So lassen sich alle Permutationen der Reihenfolge der Selektionsstati erzeugen.

Beispiel

In Quelltext 3.6 auf der nächsten Seite ist die Spezifikation der Methode *addEdge* im Metaprodukt dargestellt, wenn die beschriebenen Regeln angewendet werden. Der Umfang der Spezifikation wird daher deutlich reduziert. Dennoch ist diese Spezifikation logisch äquivalent zu der Spezifikation, die in Quelltext 3.5 auf Seite 28 dargestellt ist, wenn das Feature-Modell aus Abbildung 2.2 auf Seite 7 zugrunde gelegt wird. Das bedeutet, die Spezifikation ist genau dann erfüllt, wenn auch die andere Spezifikation erfüllt ist.

Die Implikationen `FM.FeatureModel.base ==>` konnten entfernt werden, da alle Software-Produkte, in denen die Methode *addEdge* existiert, auch das Feature *Base* enthalten. Aufgrund der Spezifikation im Feature *MaxEdges* müssen die Vorbedingungen des Features *Base* immer erfüllt sein, unabhängig davon, ob das Feature *MaxEdges* vorhanden ist, oder nicht. In der optimierten Spezifikation des Metaproduktes ist daher die dunkelgrau hervorgehobene Vorbedingung nur noch einmal vorhanden.

```
1  /*@                                                                    Metaprodukt
2    @ requires (edge != null
3    @      &&   nodes.contains(edge.first)
4    @      &&   nodes.contains(edge.second));
5    @ requires FeatureModel.maxedges ==>
6    @           (MAXEDGES != null);
7    @ ensures !FeatureModel.maxedges ==>
8    @           hasEdge(edge);
9    @ ensures FeatureModel.maxedges ==>
10   @           (\old(countEdges()) < MAXEDGES ==> hasEdge(edge));
11   @*/
12 public boolean equals(Object ob) { /* ... */ }
```

Quelltext 3.6: Optimierte Spezifikation der Methode *addEdge* im Metaprodukt nach der Komposition von *Base* und *MaxEdges* mittels *Explicit Contract Refinement*

3.3 Conjunctive Contract Refinement

Bisher wurde die Kontraktkomposition im Metaprodukt nur für *Explicit Contract Refinement* umgesetzt. Bei der Entwicklung von Software-Produktlinien werden jedoch auch andere Kompositionsverfahren, wie *Conjunctive Contract Refinement*, verwendet. Beim *Conjunctive Contract Refinement* müssen in einem Software-Produkt alle Vor- und Nachbedingungen der komponierten Features erfüllt werden [TSK+12, Ben12, TBA+13]. Beim Metaprodukt muss beachtet werden, dass nicht immer alle Features vorhanden sind. Die Vor- und Nachbedingungen eines Features müssen nur dann erfüllt sein, wenn es in der Konfiguration vorhanden ist.

3.3.1 Generierung der Spezifikationen im Metaprodukt

Wie auch beim *Explicit Contract Refinement* eignen sich Implikationen, um sicherzustellen, dass Klauseln nur gelten, wenn die Konfiguration die entsprechenden Features enthält. Wenn ein Feature vorhanden ist, müssen auch die entsprechenden Vor- und Nachbedingungen erfüllt werden. Ist das Feature hingegen nicht vorhanden, ist die Implikation wahr und die generierte Bedingung somit erfüllt.

```
1  /*@                                                                      Base
2    @ requires ob != null;
3    @ ensures \result ==> ob instanceof Edge;
4    @*/
5  public boolean equals(Object ob) { /* ... */ }
```

```
6  /*@                                                                  Weighted
7    @ requires weight != null;
8    @ ensures \result ==> weight == ((Edge)ob).weight;
9    @*/
10 public boolean equals(Object ob) { /* ... */ }
```

Quelltext 3.7: Spezifikationen der Methode *equals* in den Features *Base* und *Weighted*

In Quelltext 3.7 wird die Spezifikationen der Methode *equals* der Klasse *Edge* in den Features *Base* und *Weighted* dargestellt. Im Feature *Base* wird spezifiziert, dass

ein Objekt `ob` zur Kante äquivalent ist, wenn es eine Instanz der Klasse *Edge* ist. Der Aufrufer der Methode muss dafür sicherstellen, dass `ob` initialisiert wurde. Im Feature *Weighted* muss ein Kantengewicht definiert sein. Wenn dies der Fall ist, sind zwei Kanten äquivalent, wenn deren Kantengewichte übereinstimmen.

Bei der Komposition der beiden Features für die Generierung eines Software-Produktes würde die in Quelltext 3.8 dargestellte Spezifikation erzeugt werden. Es müssen die Vorbedingungen beider Features erfüllt werden.

Base • Weighted

```
1  /*@
2    @ requires ob != null;
3    @ requires weight != null;
4    @ ensures \result ==> ob instanceof Edge;
5    @ ensures \result ==> weight == ((Edge)ob).weight;
6    @*/
7  public boolean equals(Object ob) { /* ... */ }
```

Quelltext 3.8: Spezifikation der Methode *equals* nach der Komposition von *Base* und *Weighted* für ein Software-Produkt mittels *Conjunctive Contract Refinement*

Welche Spezifikationen im Metaprodukt erfüllt sein müssen, ist abhängig von den in der Konfiguration vorhandenen Features. Die zu einem Feature gehörenden Vor- und Nachbedingungen einer Methode müssen genau dann erfüllt werden, wenn das Feature auch vorhanden ist. Um dies zu gewährleisten, können analog zum *Explicit Contract Refinement* Implikationen eingefügt werden. Ist ein Feature in der Konfiguration vorhanden, führt dies dazu, dass die zu dem Feature gehörenden Vor- und Nachbedingungen erfüllt sein müssen. Wenn das Feature nicht vorhanden ist, ist die so entstehende Klausel immer erfüllt, da $(false \Rightarrow x) \equiv true$.

Metaprodukt

```
1  /*@
2    @ requires FeatureModel.base ==>
3    @           ob != null;
4    @ requires FeatureModel.weighted ==>
5    @           weight != null;
6    @ ensures  FeatureModel.base ==>
7    @           \result ==> ob instanceof Edge;
8    @ ensures FeatureModel.weighted ==>
9    @           \result ==> weight == ((Edge)ob).weight;
10   @*/
11 public boolean equals(Object ob) { /* ... */ }
```

Quelltext 3.9: Spezifikation der Methode *equals* im Metaprodukt nach der Komposition von *Base* und *Weighted* mittels *Conjunctive Contract Refinement*

Werden die in Quelltext 3.7 auf der vorherigen Seite dargestellten Spezifikationen bei der Metaprodukt-Generierung komponiert, wird die in Quelltext 3.9 abgebildete Spezifikation erzeugt. Wenn das Feature *Base* vorhanden ist, müssen die dunkelgrau hervorgehobenen Bedingungen erfüllt sein. Ist das Feature *Weighted* vorhanden, müssen die hellgrau hervorgehobenen Bedingungen erfüllt sein.

3.3.2 Vereinfachung der Spezifikationen

Wie beim *Explicit Contract Refinement* können Vereinfachungen vorgenommen werden, wenn das Feature-Modell betrachtet wird. Analog zum *Explicit Contract Refinement* können die Selektionsstati von Methoden-Kern-Features durch `true` ersetzt werden. Die so entstehenden Implikationen der Form $true \Rightarrow x$ können dann wiederum durch x ersetzt werden, wobei x eine Vor- oder Nachbedingung repräsentiert.

Beispiel

Das so vereinfachte Metaprodukt zu Quelltext 3.9 wird in Quelltext 3.10 dargestellt. Die dunkelgrau hinterlegten Bedingungen müssen immer gelten, da diese im Feature*Base* spezifiziert wurden und das Feature Base ein Methoden-Kern-Feature ist. Die Implikation `FM.FeatureModel.base ==>` kann daher ebenfalls entfernt werden.

Metaprodukt

```
1 /*@
2   @ requires ob != null;
3   @ requires FeatureModel.weighted ==>
4   @           weight != null;
5   @ ensures  \result ==> ob instanceof Edge;
6   @ ensures FeatureModel.weighted ==>
7   @           \result ==> weight == ((Edge)ob).weight;
8   @*/
9 public boolean equals(Object ob) { /* ... */ }
```

Quelltext 3.10: Optimierte Spezifikation der Methode *equals* im Metaprodukt nach der Komposition von *Base* und *Weighted* mittels *Conjunctive Contract Refinement*

3.4 Consecutive Contract Refinement

Bei der Komposition von Spezifikationen mittels *Consecutive Contract Refinement* müssen die zusammengehörenden Vor- und Nachbedingungen von mindestens einem vorhandenen Feature erfüllt werden [Ben12, TBA+13].

3.4.1 Generierung der Spezifikationen im Metaprodukt

Bei der Komposition der in Quelltext 3.7 auf Seite 31 dargestellten Spezifikationen mittels *Consecutive Contract Refinement* würde die in Quelltext 3.11 auf der nächsten Seite abgebildete Spezifikation in einem Software-Produkt entstehen. Die dunkelgrau hinterlegten Bedingungen stammen aus dem Feature *Base*, die hellgrau hinterlegten aus *Weighted*. Es muss die im Feature *Base* oder die in *Weighted* definierte Vorbedingung erfüllt werden. Wurde die Vorbedingung eines Features erfüllt, dann muss auch die dazugehörige Nachbedingung erfüllt werden. Dies kann realisiert werden, indem die Vorbedingung die Nachbedingung impliziert. Wenn eine Vorbedingung erfüllt war, muss somit auch die Nachbedingung erfüllt sein. Da durch die Ausführung der spezifizierten Methode Klassenvariablen oder andere Vergleichswerte so geändert werden könnten, dass die Vorbedingung nicht mehr erfüllt ist, wird mithilfe des Schlüsselwortes `old` auf den Zustand vor der Ausführung der Methode zurück gegriffen.

```
1 /*@                                                          Base • Weighted
2   @ requires (ob != null)
3   @      ||  (weight != null);
4   @ ensures \old(obj != null) ==>
5   @           \result ==> ob instanceof Edge;
6   @ ensures \old(weight != null) ==>
7   @           \result ==> weight == ((Edge)ob).weight;
8   @*/
9 public boolean equals(Object ob) { /* ... */ }
```

Quelltext 3.11: Spezifikation der Methode *equals* nach der Komposition von *Base* und *Weighted* für ein Software-Produkt mittels *Consecutive Contract Refinement*

Bei der Erzeugung des Metaproduktes kann für die Vorbedingungen nicht mit Implikationen gearbeitet werden, wie beim *Explicit Contract Refinement* oder *Conjunctive Contract Refinement*. Aufgrund der Disjunktion der Vorbedingungen muss sichergestellt werden, dass erfüllte Vorbedingungen eines Features, das in der Konfiguration nicht vorhanden ist, nicht dazu führen, dass die Vorbedingung des Metaproduktes erfüllt ist. Ebenso darf die Vorbedingung nicht erfüllt sein, sobald eines der Features nicht vorhanden ist. Daher eignet sich die Konjunktion der Vorbedingungen eines Features mit dem Selektionsstatus des Features, um die Variabilität zu kodieren.

Da die Garantien in den Nachbedingungen nur vorhanden sind, wenn das entsprechende Feature vorhanden ist, müssen auch diese angepasst werden. Hierfür sind zwei Vorgehen möglich. Zum einen kann eine Implikation analog zum *Explicit Contract Refinement* und *Conjunctive Contract Refinement* eingefügt werden. Zum anderen kann auch eine Konjunktion der Vorbedingungen mit dem Selektionsstatus des Features erstellt werden.

Wenn eines der komponierten Features keine Vorbedingungen enthält, gelten die Vorbedingungen als erfüllt. In diesem Fall müssen die Nachbedingungen immer erfüllt werden, wenn das Feature gewählt wurde. Wenn keine Vorbedingungen vorhanden sind, könnte dies in der Konjunktion in den Nachbedingungen im Metaprodukt durch `true` ersetzt werden. Da das Ergebnis der Konjunktion dann immer dem Selektionsstatus entspricht, ist es ausreichend die Konjunktion durch den Selektionsstatus zu ersetzen.

Wird die Konjunktion verwendet, ergibt sich die in Quelltext 3.12 auf der nächsten Seite dargestellte Spezifikation im Metaprodukt bei der Komposition der beiden Spezifikationen in Quelltext 3.7 auf Seite 31. Die dunkelgrau hinterlegten Vor- und Nachbedingugen stammen von dem Feature *Base*, die hellgrau hinterlegten Bedingungen von *Weighted*.

Wenn das Feature *Base* in der Konfiguration vorhanden ist und die in *Base* spezifizierten Vorbedingungen (hellgrau hervorgehoben) erfüllt sind, ist auch die Vorbedingung des Metaproduktes erfüllt. Ebenso kann auch das Feature *Weighted* vorhanden und die dort spezifizierten Vorbedingungen (dunkelgrau hervorgehoben) erfüllt sein. Es ist ausreichend, wenn einer dieser Fälle eintritt. Je nachdem, welche Vorbedingung erfüllt war, muss durch die Methode auch die Nachbedingungen des Features erfüllen, in dem die erfüllte Vorbedingung definiert wurde.

Metaprodukt

```
1 /*@
2   @ requires (FeatureModel.base && (ob != null))
3   @     ||   (FeatureModel.weighted && (weight != null));
4   @ ensures (FeatureModel.base && \old(obj != null)) ==>
5   @           \result ==> ob instanceof Edge;
6   @ ensures (FeatureModel.weighted && \old(weight != null)) ==>
7   @           \result ==> weight == ((Edge)ob).weight;
8   @*/
9 public boolean equals(Object ob) { /* ... */ }
```

Quelltext 3.12: Spezifikation der Methode *equals* im Metaprodukt nach der Komposition von *Base* und *Weighted* mittels *Consecutive Contract Refinement*

3.4.2 Vereinfachung der Spezifikationen

Wie bei den vorherigen Verfahren ist es auch hier möglich, die generierten Klauseln zu vereinfachen. Eine Vereinfachung der so erzeugten Spezifikation ist möglich, wenn die Spezifikation Selektionsstati von Methoden-Kern-Features enthält. In diesem Fall würde der Selektionsstatus immer den Wert `wahr` annehmen. Die Konjunktionen mit den Vorbedingungen hätten dann die Form `wahr && <Bedingung>`. Da gilt, dass $(wahr \wedge x) \equiv x$, kann der Selektionsstatus aus der Konjunktion entfernt werden.

Beispiel

In Quelltext 3.13 ist die so vereinfachte Spezifikation im Metaprodukt dargestellt. Da *Base* ein Methoden-Kern-Feature ist, kann dessen Selektionsstatus in der Konjunktion durch `wahr` ersetzt und daher entfernt werden. Die Nachbedingungen der beiden Features müssen erfüllt sein, wenn das entsprechende Feature in der Konfiguration vorhanden war und die in dem Feature definierte Vorbedingung vor der Ausführung der Methode erfüllt war. Ist das Feature nicht gewählt, ist die Konjunktion aus Selektionsstatus und Vorbedingung immer `falsch`. Die entsprechende Klausel ist durch die Implikation dann immer erfüllt. Ist das Feature zwar in der Konfiguration enthalten, die Vorbedingung aber nicht erfüllt, ist die Konjunktion ebenfalls `falsch`, wodurch die Klausel erfüllt wird.

Metaprodukt

```
1  /*@
2    @ requires FeatureModel.valid();
3    @ requires (ob != null)
4    @     ||   (FeatureModel.weighted && (weight != null));
5    @ ensures \old(obj != null) ==>
6    @           \result ==> ob instanceof Edge;
7    @ ensures (FeatureModel.weighted && \old(weight != null)) ==>
8    @           \result ==> weight == ((Edge)ob).weight;
9    @*/
10 public boolean equals(Object ob) { /* ... */ }
```

Quelltext 3.13: Optimierte Spezifikation der Methode *equals* im Metaprodukt nach der Komposition von *Base* und *Weighted* mittels *Consecutive Contract Refinement*

3.5 Cumulative Contract Refinement

Beim *Cumulative Contract Refinement* müssen, wie beim *Consecutive Contract Refinement* die Vorbedingungen von mindestens einem Feature erfüllt werden. Es müssen allerdings alle Nachbedingungen aller Features wie beim *Conjunctive Contract Refinement* erfüllt werden.

Bei der Metaprodukterzeugung kann daher jeweils genauso vorgegangen werden, wie bei den anderen beiden Verfahren. Dies gilt auch für die dort beschriebenen Vereinfachungen. Die Vorbedingungen werden analog zu den Vorbedingungen beim *Consecutive Contract Refinement* komponiert. Das heißt, die Vorbedingungen werden mit dem Selektionsstatus des entsprechenden Features konjugiert. Anschließend werden diese Konjunktionen der einzelnen Features in einer Disjunktion im Metaprodukt zusammengeführt. Bei den Nachbedingungen wird analog zum *Conjunctive Contract Refinement* eine Implikation eingefügt.

Werden die beiden Spezifikationen in Quelltext 3.7 auf Seite 31 mittels *Cumulative Contract Refinement* kombiniert ergibt sich daher im Metaprodukt die in Quelltext 3.14 dargestellte Spezifikation. Wie beim *Consecutive Contract Refinement* sind die Vorbedingungen durch eine Disjunktion verknüpft. Bei der dunkelgrau hinterlegten Vorbedingung des Features *Base* konnte die Implikation entfernt werden, da das Feature *Base* ein Methoden-Kern-Feature für *equals* ist.

Metaprodukt

```
1 /*@
2   @ requires (ob != null)
3   @      ||   (FeatureModel.weighted && (weight != null));
4   @ ensures \result ==> ob instanceof Edge;
5   @ ensures FeatureModel.weighted ==>
6   @           \result ==> weight == ((Edge)ob).weight;
7   @*/
8 public boolean equals(Object ob)
```

Quelltext 3.14: Optimierte Spezifikation der Methode *equals* im Metaprodukt nach der Komposition von *Base* und *Weighted* mittels *Cumulative Contract Refinement*

3.6 Plain Contracting

Beim *Plain Contracting* wird eine Spezifikation bei der Einführung einer Methode definiert. Bei der Komposition mit weiteren Features wird diese Spezifikation nicht verändert.

3.6.1 Allgemeine Generierung der Spezifikation

Auf den ersten Blick scheint die Kontrakt-Komposition mittels *Plain Contracting* bei der Metaprodukt-Generierung trivial zu sein. Es könnte das gleiche Vorgehen verwendet werden, wie auch bei der Erstellung eines Software-Produktes, da später komponierte Features keine Änderungen an den Spezifikationen vornehmen können.

Problematisch ist aber, dass eine Methode auch durch mehr als ein Feature eingeführt werden kann. Es könnten zum Beispiel zwei alternative Features die gleiche Methode mit unterschiedlichen Spezifikationen einführen. Bei einem normalen

Software-Produkt würde dann die Spezifikation des gewählten Features gelten. Die Bedingungen im Metaprodukt dürfen daher nur gelten, wenn das Feature auch vorhanden ist. Hierfür ist wiederum eine Implikation, wie beim *Conjunctive Contract Refinement* möglich.

Würden allerdings bei jeder Komposition von zwei Features die Vor- und Nachbedingungen beider Features verwendet werden, könnte das Metaprodukt Spezifikationen enthalten, die in keinem Produkt vorhanden sind. Diese Spezifikationen müssten bei der Software-Produkt-Generierung immer bereits vorhandene Spezifikationen überschreiben. Dies könnte zum Beispiel der Fall sein, wenn ein Entwickler neue Spezifikationen für eine Methode in einem Feature definiert, das die Methode nicht einführt. Bei der Komposition dürfen daher nur Features betrachtet werden, die die Methode einführen. Sobald ein Methoden-Kern-Feature komponiert wurde, ist keine Änderung der Kontrakte mehr möglich. Wenn die Methode aber in mehreren alternativen Features eingeführt wird, ist nicht zwingend ein für die Methode obligatorisches Feature vorhanden. In diesem Fall muss geprüft werden, ob das Feature, das komponiert werden soll, eines der bereits komponierten Features voraussetzt. Wenn dies zutrifft, können die Spezifikationen des Features nicht hinzugefügt werden.

Im Gegensatz zu den anderen bisher beschriebenen Verfahren, ist beim *Plain Contracting* für die Erstellung des Metaproduktes die Kenntnis des Feature-Modells wichtig. Ohne diese Informationen sind bei den anderen Verfahren lediglich einige Vereinfachungen der erzeugten Spezifikation nicht möglich. Beim *Plain Contracting* kann ohne die Informationen über das Feature-Modell kein korrektes Metaprodukt erzeugt werden. Wenn die Informationen zum Feature-Modell nicht vorhanden sind, muss entweder angenommen werden, dass eine Methode nur von genau einem Feature eingeführt werden kann, oder dass alle Features eine Methode einführen können. Für beide Annahmen gibt es Software-Produktlinien, auf die das Vorgehen nicht anwendbar ist.

Würden die beiden Spezifikationen in Quelltext 3.7 auf Seite 31 für das Metaprodukt mittels *Plain Contracting* komponiert werden, ergibt sich daher die in Quelltext 3.15 dargestellte Spezifikation der Methode *equals*. Da das Feature *Base* ein Methoden-Kern-Feature ist und die Methode einführt, gelten im Metaprodukt auch die Vor- und Nachbedingungen des Features *Base*. Das Feature *Weighted* verfeinert die Methode lediglich. Daher sind bei der Komposition von *Weighted* bereits Spezifikationen vorhanden. Kein Software-Produkt kann die, in *Weighted* definierten Spezifikationen enthalten.

Metaprodukt

```
1 /*@
2   @ requires FeatureModel.base ==>
3   @          (ob != null);
4   @ ensures FeatureModel.base ==>
5   @          (\result ==> ob instanceof Edge);
6   @*/
7 public boolean equals(Object ob)
```

Quelltext 3.15: Spezifikation der Methode *equals* der Klasse *Edge* im Metaprodukt nach der Komposition der Features *Base* und *Weighted* mittels *Plain Contracting*

3.6.2 Vereinfachung der Spezifikation

Wie auch bei den vorherigen Verfahren ist eine Vereinfachung der so erzeugten Spezifikation möglich. Die Implikation kann entfernt werden, wenn es sich bei dem Feature um ein Methoden-Kern-Feature handelt.

Da *Base* ein Methoden-Kern-Feature ist, kann bei dem Beispiel die Implikation entfernt werden. Die so entstehende vereinfachte Spezifikation in Quelltext 3.16 ist identisch zu der Spezifikation aus dem Feature *Base*. Lediglich die in Abschnitt 3.1.3 beschriebenen zusätzlichen Vorbedingungen würden noch eingefügt werden.

Metaprodukt

```
1 /*@
2   @ requires ob != null;
3   @ ensures \result ==> ob instanceof Edge;
4   @*/
5 public boolean equals(Object ob)
```

Quelltext 3.16: Optimierte Spezifikation der Methode *equals* der Klasse *Edge* im Metaprodukt nach der Komposition der Features *Base* und *Weighted* mittels *Plain Contracting*

3.7 Method-Based Contract Refinement

Beim *Method-Based Contract Refinement* wird der Kompositionsmechanismus für jede Methode individuell festgelegt. Außerdem kann auch ein Wechsel des Kompositionsverfahrens einer Methode definiert werden [Wei13].

3.7.1 Generierung der Spezifikation im Metaprodukt

Bei der Metaproduktgenerierung muss daher zunächst das zu verwendende Verfahren ermittelt werden. Dann können die Regeln des entsprechenden Verfahrens angewendet werden.

Weigelt beschreibt jedoch auch einen möglichen Wechsel des Kompositionsverfahrens [Wei13]. Der Wechsel des Kompositionsverfahrens erfolgt nach der Komposition des Features, in dem das neue Verfahren definiert wurde. Bei jedem möglichen Wechsel des Kompositionsverfahrens entstehen zwei Spezifikationen, die getrennt betrachtet werden müssen. Die erste Spezifikation beschreibt den Fall, dass das Feature in der Konfiguration nicht vorhanden ist. Bei der Komposition mit dem nächsten Feature, erfolgt diese weiterhin mit dem alten Kompositionsverfahren. Die zweite Spezifikation beschreibt den Fall, dass das Feature in der Konfiguration vorhanden ist und somit ein Wechsel des Kompositionsverfahrens stattfindet.

Das so entstehende Gebilde kann durch einen Baum repräsentiert werden. In Abbildung 3.1 auf Seite 40 ist ein Beispiel für einen solchen Baum abgebildet. Die Blätter des Baumes stellen dabei die Spezifikationen und das zu verwendende Verfahren dar. Erfolgt die Komposition mit einem weiteren Feature, muss dieses mit jedem einzelnen Blatt separat komponiert werden. Wenn es durch die Komposition mit einem Blatt zu einem Wechsel des Kompositionsverfahrens kommen kann, wird dieses Blatt in einen Knoten mit zwei neuen Blättern umgewandelt. Diese beiden Blätter

repräsentieren dann die zwei neuen Spezifikationen. In einer Spezifikation wurde das Kompositionsverfahren gewechselt. Das zweite Blatt enthält die alte Spezifikation, wobei diese nur dann gilt, wenn das neue Feature nicht vorhanden ist.

Bei der Erstellung eines solchen Baumes muss zunächst ein Startknoten festgelegt werden. Dafür muss ein Knoten mit dem Verfahren *Explicit Contract Refinement* ohne Vor- oder Nachbedingungen gewählt werden. Die erste, bei der Erstellung des Metaproduktes betrachtete Spezifikation kann nicht als Wurzelknoten verwendet werden. Wenn die Methode durch mehr als ein Feature eingeführt wird, wäre dies sonst nicht abbildbar, da der Fall, dass das erste betrachtete Feature nicht vorhanden ist, nicht beachtet wird.

Anhand eines Beispiels soll die Erstellung eines solchen Baumes näher erläutert werden. Betrachtet wird eine Software-Produktlinie mit drei Features A, B und C. Diese enthalten die in Quelltext 3.17 dargestellten Spezifikationen der Methode *method*. Die einzelnen Vorbedingungen `rA`, `rB` und `rC` sowie die Nachbedingungen `eA`, `eB` und `eC` repräsentieren beliebige Vor- und Nachbedingungen. Alle drei Features sind beliebig kombinierbar. Die Komposition erfolgt in der Reihenfolge A, B, C.

```
1  /*@ \conjunctive_contract                                    A
2    @ requires rA;
3    @ ensures eA;
4    @*/
5  public void method()
```

```
6  /*@ \cumulative_contract                                     B
7    @ requires rB;
8    @ ensures eB;
9    @*/
10 public void method()
```

```
11 /*@ \cumulative_contract                                     C
12   @ requires rC;
13   @ ensures eC;
14   @*/
15 public void method()
```

Quelltext 3.17: Spezifikation der Methode *method* in den Features A, B und C

Die Erstellung des Baumes beginnt in einem Wurzelknoten mit dem Kompositionsverfahren *Explicit Contract Refinement* ohne Vor- und Nachbedingungen. Bei der Komposition dieses Knotens mit der Spezifikation aus Feature A würde ein Wechsel des Kompositionsverfahrens von *Explicit Contract Refinement* zu *Conjunctive Contract Refinement* stattfinden. Daher werden zwei neue Blätter erstellt. Das erste Blatt enthält den unveränderten Wurzelknoten, das zweite Blatt die komponierte Spezifikation. Anschließend wird das Feature B hinzugefügt. Dabei findet in beiden Blättern ein Kompositionswechsel statt, sodass diese sich wiederum verzweigen. In Abbildung 3.1 auf der nächsten Seite wird der so entstandene Baum abgebildet.

In der ersten Zeile der Knoten und Blätter wird jeweils das Kompositionsverfahren angegeben. Darunter folgen die Vorbedingungen und danach die Nachbedingungen. Da nur das Feature A hinzugefügt wurde, ist nur ein Blatt mit Vor- und Nachbedingungen enthalten. An den Ästen wurde dabei markiert, ob das Feature A hinzugefügt

wurde (`A`) oder nicht (`!A`). Ist das Feature A vorhanden, wird als neues Kompositionsverfahren *Conjunctive Contract Refinement* verwendet. Ist das Feature nicht vorhanden, bleibt es beim Kompositionsverfahren *Explicit Contract Refinement*.

Beim Hinzufügen des Features B müssen die beiden so entstandenen Blätter wieder aufgeteilt werden, da wiederum jeweils ein potentieller Kompositionswechsel stattfindet. Die so entstehenden vier Spezifikationen bilden alle vier möglichen Kombinationen der beiden Features ab. Welche Kombination durch eine Spezifikation abgebildet wird, kann ermittelt werden, indem der Weg zum Wurzelknoten betrachtet wird. An den Ästen ist jeweils notiert, ob bestimmte Features vorhanden sind oder nicht. Das Nicht-Vorhandensein wird dabei durch ein vorangestelltes „!“ dargestellt.

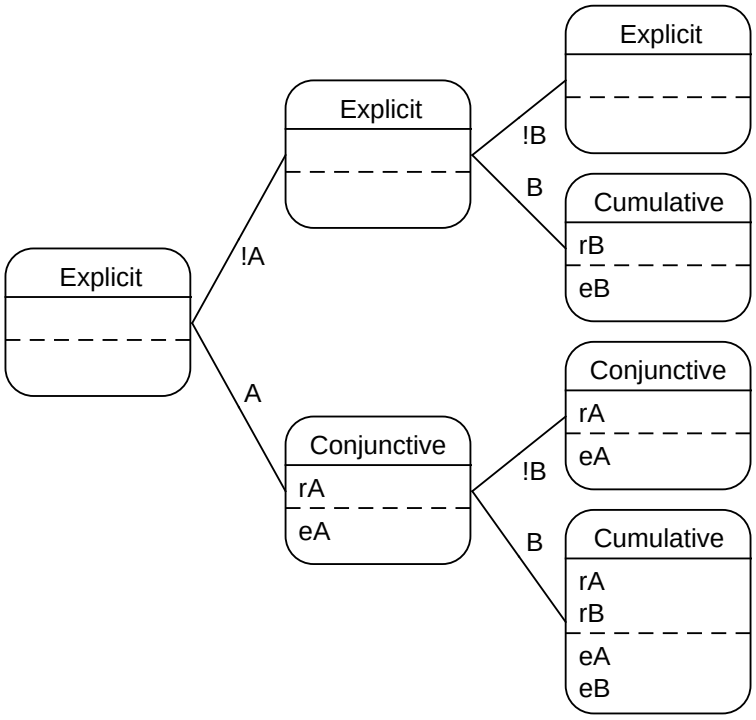

Abbildung 3.1: Baum zur Repräsentation von Spezifikationen beim *Method-Based Contract Refinement* nach dem Hinzufügen der Features A und B

Wird nun auch das Feature C hinzugefügt entsteht der in Abbildung 3.2 auf der nächsten Seite dargestellte Baum. Wenn zuvor das Feature B vorhanden war, erfolgt kein Wechsel des Kompositionsverfahrens. In diesem Fall kann die Komposition mit dem entsprechenden Verfahren erfolgen. Die Komposition von Spezifikationen im Metaprodukt mittels *Cumulative Contract Refinement* bildet bereits ab, ob ein Feature vorhanden ist, oder nicht. Im Baum wird dies mit `cum()` dargestellt. Die Funktion *cum* repräsentiert die Komposition der Kontrakte mittel *Cumulative Contract Refinement*.

Bei den beiden Spezifikationen, bei denen das Feature B zuvor nicht komponiert wurde, findet ein Wechsel des Kompositionsverfahrens statt. Daher müssen die Blätter dieser Spezifikationen wiederum aufgeteilt werden. Der so entstehende Baum enthält

sechs Blätter, die jeweils die Spezifikation für bestimmte Feature-Kombinationen repräsentieren. Das Metaprodukt muss diese sechs Spezifikationen abbilden.

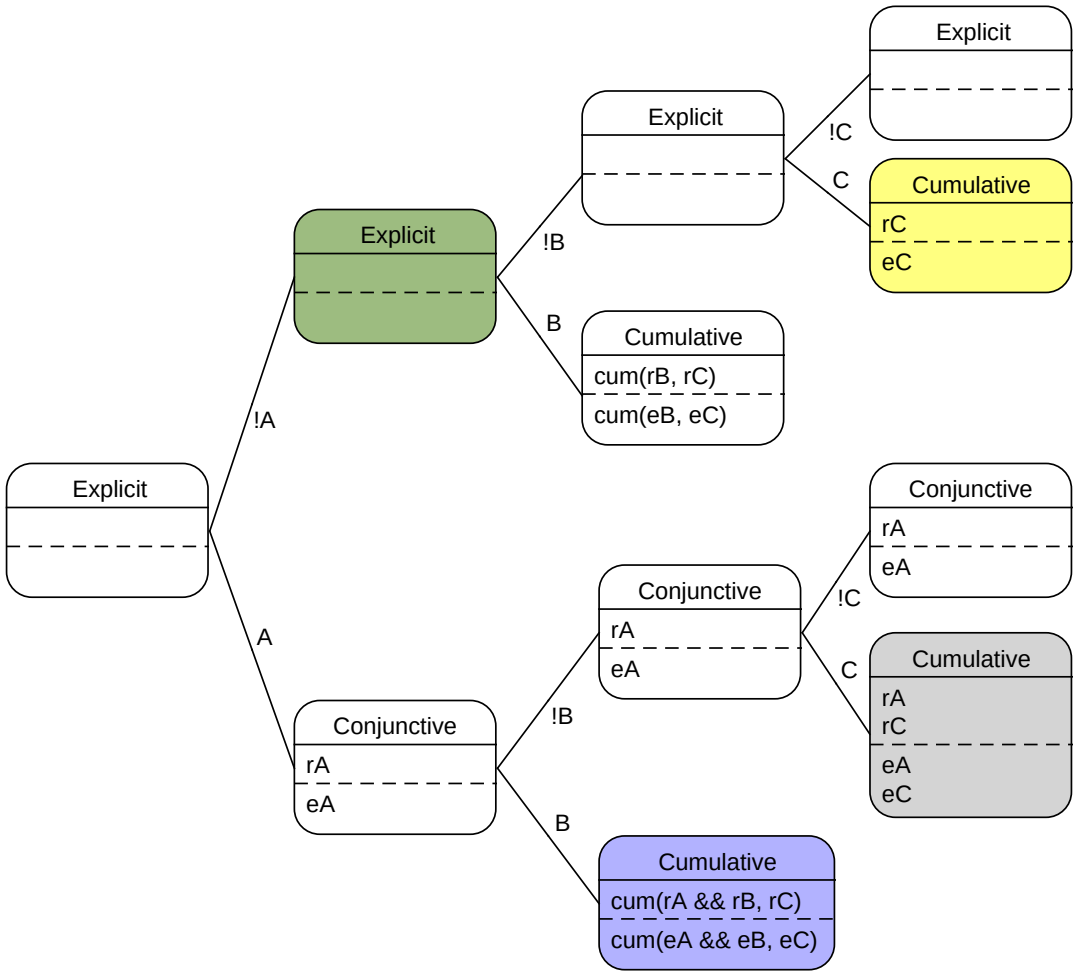

Abbildung 3.2: Baum zur Repräsentation von Spezifikationen beim *Method Based Contract Composition* nach dem Hinzufügen der Features A, B und C

Um sicherzustellen, dass eine Spezifikation nur gilt, wenn auch die entsprechenden Features vorhanden sind, können Implikationen verwendet werden. Vor jede Vor- und Nachbedingung eines Blattes des Baumes müssen Implikationen aus den an den Ästen zwischen dem Wurzelknoten und dem Blatt angegebenen Selektionsstati eingefügt werden. Damit die Spezifikationen in dem, in Abbildung 3.2 gelb hervorgehobenen Blatt gelten, dürfen beispielsweise die Features A und B nicht in der Konfiguration vorhanden sein. C muss jedoch selektiert sein. Das Metaprodukt muss daher für dieses Blatt die in Quelltext 3.18 auf der nächsten Seite dargestellten Spezifikationen enthalten.

Um das Metaprodukt zu erstellen, müssen also alle Blätter betrachtet werden. Vor jede Bedingung in den Spezifikationen muss die entsprechende Implikationskette aus den Selektionsstati der Features eingefügt werden. Die so entstandenen Bedingungen können in das Metaprodukt eingefügt werden. Durch die Verwendung der Implika-

tionen gelten nur die Bedingungen, die in einem Software-Produkt mit der gleichen Konfiguration vorhanden wären.

Metaprodukt

```
1 /*@
2   @ requires !FeatureModel.A ==> !FeatureModel.B ==>
3   @              FeatureModel.C ==> rC;
4   @ ...
5   @ ensures !FeatureModel.A ==> !FeatureModel.B ==>
6   @              FeatureModel.C ==> eC;
7   @ ...
8   @*/
9 public void method()
```

Quelltext 3.18: Ausschnitt aus dem Metaprodukt nach der Komposition mithilfe von *Method-Based Contract Refinement*

3.7.2 Vereinfachung der Spezifikation

Bereits bei drei Features entsteht so ein sehr umfangreicher Baum. Dieser lässt sich aber reduzieren, wenn Wissen über das Feature-Modell vorhanden ist.

Pflicht-Features

Bei Methoden-Kern-Features, muss das Blatt, welches die Spezifikation repräsentiert, wenn das Feature nicht vorhanden ist, nicht betrachtet werden. Es kann aus dem Baum gestrichen beziehungsweise gar nicht erst eingefügt werden. Angenommen, bei der Erstellung des Baumes in Abbildung 3.2 auf der vorherigen Seite ist bekannt, dass das Feature A obligtorisch ist. Dann kann der grün hervorgehobene Knoten mit allen Kindern aus dem Baum entfernt werden.

Ungültige Feature-Kombinationen

Weiterhin ist es möglich, dass Blätter für Feature-Kombinationen entstehen, die nach dem Feature-Modell nicht erlaubt sind. In diesem Fall können diese Blätter beziehungsweise Knoten ebenfalls entfernt werden. Auch wenn kein Wechsel des Kompositionsverfahrens erfolgt, können mögliche invalide Konfigurationen beachtet werden. Die Komposition eines Blattes mit dem Feature muss dann nicht mehr ausgeführt werden.

Wenn beispielsweise die Feature-Kombination aus A und C nicht erlaubt ist, kann der Baum in Abbildung 3.2 auf der vorherigen Seite vereinfacht werden. In diesem Fall kann der grau hinterlegte Knoten entfernt werden. Außerdem ist eine Komposition der ursprünglichen Spezifikation des blau hervorgehobenen Knotens mit der Spezifikation des Features C nicht mehr nötig. Der Knoten behält in diesem Fall seine ursprüngliche Spezifikation (siehe Abbildung 3.1 auf Seite 40).

Optimierungen nach Kompositionsverfahren

Die Spezifikationen der einzelnen Blätter des Baumes können mit den bereits beschriebenen Kompositions-Regeln des entsprechenden Verfahrens erstellt werden.

Dementsprechend können auch die dort beschriebenen Vereinfachungen genutzt werden. Weiterhin können bei der Komposition die selektierten beziehungsweise deselektierten Features an den Ästen als festgelegt betrachtet werden.

Angenommen, das Feature C wird durch das Feature A impliziert. Dann lassen sich die Spezifikationen im blau und im grau hervorgehobenen Knoten vereinfachen. Um dies zu ermöglichen, kann bei der Komposition das Feature C als Methoden-Kern-Feature betrachtet werden. Beide Knoten repräsentieren nur Produkte, in denen das Feature A vorhanden ist. Durch die Implikation müssen alle simulierten Software-Produkte auch das Feature C enthalten.

3.8 Zusammenfassung

Die Komposition von Kontrakten bei der Erzeugung eines Metaproduktes wurde um zusätzliche Kompositionstechniken erweitert. Neben *Explicit Contract Refinement* wurden die Kompositionsverfahren *Conjunctive Contract Refinement*, *Consecutive Contract Refinement*, *Cumulative Contract Refinement*, *Plain Contracting* und *Method-Based Contract Refinement* betrachtet. Weiterhin wurden Optimierungen an den erzeugten Spezifikationen vorgeschlagen. Durch diese Optimierungen wird die Lesbarkeit der Spezifikationen im Metaprodukt verbessert. Bisher wurde jedoch nicht geprüft, ob die Regeln für die Komposition und Optimierung umsetzbar sind. Weiterhin ist nicht klar, ob beziehungsweise welchen Vorteil diese Optimierungen bei der Analyse des Metaproduktes mit Analysetools wie *Java Pathfinder* oder *Monkey* erzielen.

4. Toolunterstützung zur optimierten Kontrakt-Komposition

Nachdem in Kapitel 3 die Regeln für die Erstellung des Metaproduktes vorgestellt wurden, folgt in diesem Kapitel die prototypische Umsetzung dieser Regeln. Für die Komposition der Feature-Module wird *FeatureHouse* [AKL13] verwendet. Für die Entwicklung von Software-Produktlinien wird das Eclipse-Plugin *FeatureIDE* [TKB+14] genutzt. Diese beiden Tools werden erweitert, um die zusätzlichen Kompositionsverfahren bei der Metaprodukt-Generierung sowie die Optimierung der erzeugten Spezifikationen zu unterstützen.

In Abschnitt 4.1 wird zunächst die Erweiterung von *FeatureHouse* betrachtet. Dazu wird zunächst in Abschnitt 4.1.1 ein neu erstelltes Interface zur Kommunikation mit *FeatureIDE* beschrieben. In Abschnitt 4.1.2 folgt die Beschreibung der Implementierung der Kontraktkomposition. Die Erweiterung von *FeatureIDE* wird in Abschnitt 4.2 betrachtet.

4.1 Erweiterung von FeatureHouse

Mit *FeatureHouse* ist die sprachunbahängige Komposition von Feature-orientierten Programen möglich [AKL13]. Ausgehend von der Grammatik einer Sprache erzeugt *FeatureHouse Feature Structure Trees* für die Feature-Module. Diese Bäume können dann komponiert werden. Dafür müssen Methoden für die Kompositon der einzelnen Elemente definiert werden. Durch Benduhn [Ben12] und Weigelt [Wei13] wurde *FeatureHouse* bereits erweitert, um die Komposition von JML-Spezifikationen zu ermöglichen.

Aufbauend auf der Arbeit von Benduhn hat Meinicke die Kontrakt-Komposition mittels *Explicit Contract Refinement* für die Metaprodukt-Generierung in *FeatureHouse* implementiert [Mei13]. Diese Implementierung wird verwendet und erweitert.

Da in *FeatureHouse* keine Informationen über das Feature-Modell vorhanden sind, wurde eine Schnittstelle definiert, um diese aus *FeatureIDE* bereitzustellen. Für diese Schnittstelle wurde ein Interface definiert, das in *FeatureIDE* implementiert wurde. Außerdem wird bisher lediglich die Kontraktkomposition mittels *Explicit Contract Refinement* bei der Metaprodukt-Generierung unterstützt. Die Methoden für die Komposition mithilfe der anderen Verfahren wurden daher implementiert.

4.1.1 Interface FeatureModelInfo

Um die Vereinfachungen zu ermöglichen, sind Kenntnisse über das Feature-Modell nötig. Diese sind in *FeatureHouse* nicht vorhanden und müssen von dem aufrufenden Tool geliefert werden. Um dies zu ermöglichen, wurde ein Interface definiert. Dieses kann im aufrufenden Tool instanziiert werden. So kann die Metaprodukt-Generierung in beliebigen Tools verwendet werden.

Die Methoden des Interfaces lassen sich in drei Kategorien einteilen. Die erste Kategorie umfasst alle Methoden, mit denen Feature-Kombinationen auf ihre Gültigkeit überprüft werden können. Die Methoden der zweiten Kategorie dienen dazu, zu ermitteln, ob Features Kern-Features oder Methoden-Kern-Features sind. Mithilfe der dritten Gruppe übergibt *FeatureHouse* Informationen an das Interface, mit denen einige Berechnungen innerhalb der Klasse ermöglicht werden.

Methoden zur Überprüfung von Feature-Kombinationen

Um zu überprüfen, ob Feature-Kombinationen erlaubt sind, nutzt *FeatureHouse* zehn Methoden des Interfaces. In Tabelle 4.1 sind diese Methoden mit Eingabe- und Rückgabewerten dargestellt.

Methode	Eingabewerte	Rückgabewerte
selectFeature	Feature-Name (String)	-
eliminateFeature	Feature-Name (String)	-
resetSelections	-	-
resetEliminations	-	-
reset	-	-
isValidSelection	-	`boolean`
canBeSelected	Feature-Name (String)	`boolean`
canBeEliminated	Feature-Name (String)	`boolean`
isAlwaysSelected	Feature-Name (String)	`boolean`
isAlwaysEliminated	Feature-Name (String)	`boolean`

Tabelle 4.1: Methoden zur Überprüfung von Feature-Kombinationen im Interface *FeatureModelInfo*

Mithilfe der Methode *selectFeature* werden die selektierten Features einzeln festgelegt. Die Methode *eliminateFeature* dient der Festlegung der deselektierten Features.

Als Eingabe dient jeweils der Name des Features als String. Mit *resetSelections*, *resetEliminations* und *reset* werden die Selektionen, Deselektionen beziehungsweise beides wieder gelöscht.

Nachdem mit diesen Methoden die selektierten und deselektierten Features festgelegt wurden, erwartet *FeatureHouse* als Ergebnis der Methode *isValidSelection* den Wert `true`, wenn mindestens ein valides Software-Produkt bekannt ist, das diese Feature-Kombination enthält. Die Methode *canBeSelected* hat den Rückgabewert `true`, wenn das angegebene Feature selektiert werden kann. Analog dazu ist das Ergebnis der Methode *canBeEliminated* `true`, wenn das angegebene Feature deselektiert werden kann. Durch die angegebene Feature-Kombination können Features immer selektiert beziehungsweise deselektiert sein. Mithilfe der Methoden *isAlwaysSelected* und *isAlwaysRejected* kann ermittelt werden, ob dies auf das angegebene Feature zutrifft.

Methoden zur Überprüfung auf Kern-Features

Für die Optimierungen ist es nötig, zu prüfen, ob Features Kern-Features sind. Kern-Features sind Features, die in jedem Software-Produkt enthalten sind. Weiterhin ist es nötig, zu überprüfen, ob Features Methoden-Kern-Features sind. Dies sind Features, die in jedem Software-Produkt vorhanden sind, das eine bestimmte Methode enthält. Um Kern-Features und Methoden-Kern-Features zu ermitteln, werden die in Tabelle 4.2 dargestellten Methoden verwendet.

Methode	Eingabewerte	Rückgabewerte
isCoreFeature	Feature-Name (String) useSelection* (boolean)	`boolean`
isMethodCoreFeature	Klassen-Name (String) Methoden-Name (String) Feature-Name (String) useSelection* (boolean)	`boolean`

Tabelle 4.2: Methoden zur Überprüfung von Methoden-Kern-Features und Kern-Features im Interface *FeatureModelInfo* (* - optionale Parameter)

Bei der Methode *isCoreFeature* erwartet *FeatureHouse* den Wert `true`, wenn das übergebene Feature ein Kern-Feature ist. Um zu überprüfen, ob ein Feature ein Methoden-Kern-Feature ist, wird die Methode *isMethodCoreFeature* verwendet. Ist das angegebene Feature in allen Software-Produkte enthalten, in denen die übergebene Klasse mit dem angegebenen Namen existiert, ist der Rückgabewert der Methode `true`.

Beide Methoden können auch mit einem optionalen boolschen Eingabewert `useSelection` durch FeatureHouse aufgerufen werden. Wird der Wert `true` übergeben, wird die Menge der betrachteten Software-Produkte weiter eingeschränkt. Jedes Software-Produkt muss alle selektierten Features und darf keines der deselektierten Features enthalten. Dieser zusätzliche Parameter wird für die Komposition mittels *Method-Based Contract Refinement* benötigt.

Methoden für Informationen aus FeatureHouse

Da die von *FeatureHouse* erstellten Feature-Structure-Trees für die Berechnungen von Vorteil sein können, werden diese mit der Methode

```
addFeatureNodes(List<FSTNonTerminal> features)
```

mitgeteilt. Über diese Bäume kann zum Beispiel ermittelt werden, in welchen Features eine Methode implementiert wurde.

Implementierung in FeatureHouse

Um eine Möglichkeit zu bieten, das Metaprodukt auch ohne Kenntnisse über das Feature-Modell zu erzeugen, wurde eine Fallback-Klasse implementiert. Mit dieser Klasse wird ein Feature-Modell angenommen, in dem alle Feature-Kombinationen möglich sind. Weiterhin wird davon ausgegangen, dass jedes Feature optional ist. In Tabelle 4.3 sind die Rückgabewerte der Methoden des Interfaces in dieser Fallback-Klasse aufgeführt. Wenn diese Klasse verwendet wird, werden keine Optimierungen an den Spezifikationen aufgrund des Feature-Modells vorgenommen. Es muss beachtet werden, dass das Metaprodukt bei der Verwendung dieser Klasse und dem Kompositionsverfahren *Plain Contracting* nicht korrekt ist, da die Informationen aus dem Feature-Modell benötigt werden.

Methode	Rückgabewert
isCoreFeature	`false`
isMethodCoreFeature	`false`
isValidSelection	`true`
canBeSelected	`true`
canBeEliminated	`true`
isAlwaysSelected	`false`
isAlwaysEliminated	`false`

Tabelle 4.3: Rückgabewerte der Fallback-Klasse des Interfaces

4.1.2 Kontraktkomposition

Meinicke hat die in seiner Arbeit beschriebene Generierung des Metaproduktes in der Klasse *ContractCompositionMeta* im Tool *FeatureHouse* umgesetzt [Mei13]. Diese Klasse wird genutzt, um die in Kapitel 3 beschriebenen Neuerungen umzusetzen. Die beschriebenen Regeln je Verfahren wurden jeweils in entsprechenden Methoden für jedes Verfahren implementiert.

Die Vor- und Nachbedingungen einzelner Spezifikationen lassen sich mit vorhanden Methoden ermitteln. Das Ergebnis ist jeweils eine Liste der Bedingungen. Bei *Cojunctive Contract Refinement*, *Consecutive Contract Refinement* und *Cumulative Contract Refinement* können diese Klauseln dann in einer Schleife bearbeitet und

dem Ergebnis unter Verwendung der in Kapitel 3 beschriebenen Regeln hinzugefügt werden.

Beim *Plain Contracting* wird während der Komposition eine zusätzliche Vorbedingung eingefügt, wenn die Spezifikation nicht mehr verändert werden kann. Bei späteren Kompositionen kann dann überprüft werden, ob diese Vorbedingung vorhanden ist. Damit die erzeugte Spezifikation im Metaprodukt korrekt ist, wird diese Vorbedingung nach der letzten Komposition wieder aus der Spezifikation entfernt.

Wenn die Kompositon mittels *Method-Based Contract Refinement* erfolgt, ist es erforderlich, den in Abschnitt 3.7 beschriebenen Baum zur Verfügung zu haben. Die Implementierung in *FeatureHosue* sieht dies nicht vor, sodass der Baum in den Spezifikationen kodiert wurde. Es ist nicht der ganze Baum nötig. Es ist ausreichend, wenn die Blätter mit den Spezifikationen, den Selektionsstati, um zu dem Blatt zu kommen und das Kompositionsverfahren odieren. Hierfür wurden pro Kompositionsverfahren Schlüsselwörter definiert. Diese beschreiben zum einen das zu verwendende Kompositionsverfahren, zum anderen können dahinter die nötigen Selektionsstati abgelegt werden. Um die, in *FeatureHouse* vorhandenen Methoden nutzen zu können, müssen die dabei entstehenden Spezifikationen gültig sein. Um dies zu gewährleisten, werden diese Schlüsselwörter als Vor- oder Nachbedingungen eingefügt. Weiterhin müssen zunächst alle Vor- und dann alle Nachbedingungen definiert werden. Daher ist es nötig, die Schlüsselwörter sowohl bei den Vor-, als auch bei den Nachbedingungen einzufügen. Nach der Definition des Kompositionsverfahrens und der nötigen Selektionsstati folgen die Vor- beziehungsweise Nachbedingungen des Knotens.

Metaprodukt

```
1 /*@
2   @ requires FM.CompositionExplicit(!a);
3   @ requires FM.CompositionConjunctive(a);
4   @ requires rA;
5   @ ensures FM.CompositionExplicit(!a);
6   @ ensures FM.CompositionConjunctive(a);
7   @ ensures eA;
8   @*/
9 public void method()
```

Quelltext 4.1: Zwischenergebnis bei der Komposition mittels *Method-Based Contract Refinement*

In Quelltext 4.1 ist eine solche Kodierung dargestellt. Es repräsentiert das Zwischenergebnis bei der Kompositon des Beispiels aus Abschnitt 3.7 nach der Kompositon des Features *A*. `FM.CompositionExplicit()` in Zeilen 2 und 5 repräsentieren den Fall, dass das Feature nicht vorhanden ist. Da das erste Feature mit einer leeren Spezifikation mittels *Explicit Contract Refinement* komponiert werden muss, bleibt die Spezifikation leer, wenn das Feature nicht gewählt ist. Ist das Feature *A* gewählt, ist der neue Kompositionsmechanismus *Conjunctive Contract Refinement*. Dies wird durch die Klauseln in den Zeilen 3 und 6 definiert. In den Zeilen 4 und 7 folgen dann die in *A* definierten Vor- beziehungsweise Nachbedingungen.

Für die Komposition der einzelnen Knoten mit einer neuen Spezifikation kann so für jeden Knoten eine Spezifikation erstellt werden. Anschließend kann die für das jeweilige Kompositionsverfahren vorhandene Methode zur Komposition genutzt werden.

Um zu gewährleisten, dass die Methoden von den bereits bekannten Selektionsstati profitieren und die resultierende Spezifikation entsprechend vereinfacht werden kann, wurde eine weitere Klasse für das Interface *FeatureModelInfo* erstellt. Diese stellt einen Wrapper für das vom Aufrufer übergebene Objekt dar. So ist es möglich zusätzliche Features zu selektieren beziehungsweise zu deselektieren, ohne die Methoden für die anderen Kompositonsverfahren anzupassen.

Die so erzeugte Spezifikation muss vor der Ausgabe jedoch noch bearbeitet werden. Die eingefügten Schlüsselwörter müssen entfernt werden und die durch sie definierten Selektionsstati müssen als Implikation vor die dazugehörigen Bedingungen eingefügt werden. Aus Quelltext 4.1 auf der vorherigen Seite wird dann die, in Quelltext 4.2 dargestellte Spezifikation.

```
1 /*@                                                    Metaprodukt
2   @ requires FM.FeatureModel.a ==> rA;
3   @ ensures FM.FeatureModel.a ==> eA;
4   @*/
5 public void method()
```

Quelltext 4.2: Ergebnis bei der Komposition mittels *Method-Based Contract Refinement*

4.2 Erweiterung von FeatureIDE

Das Tool *FeatureIDE* ist Eclipse-PlugIn zur Feature-orientierten Softwareentwicklung [TKB+14]. Ziel von *FeatureIDE* ist es, den kompletten Entwicklungsprozess einer Software-Produktlinie zu unterstützen und eine Integrierte Entwicklungsumgebung (IDE) bereit zu stellen [TKB+14]. Die Entwicklung von *FeatureIDE* fokussiert sich auf Lehre und Forschung und stellt eine OpenSource-Alternative zu kommerziellen Tools, wie *pure::variants*[1] der Firma *pure::systems* zur Verfügung. *FeatureIDE* bietet eine gemeinsame Benutzeroberfläche für verschiedene Tools und kann Aufgaben automatisieren, für die zuvor komplexe Toolketten nötig waren [TKB+14]. *FeatureIDE* wurde erweitert, um die Anpassungen in *FeatureHouse* nutzbar zu machen.

Implementierung des Interfaces FeatureModelInfo

In *FeatureIDE* erfolgt die Implementierung einer Klasse mit dem Interface *FeatureModelInfo*. Dabei wird die bereits vorhandene Klasse *Configuration* genutzt. Diese bildet Konfigurationen ab. Jedes Feature erhält zwei Selektionsstati: einen manuell und einen automatisch festgelegten. Diese können jeweils drei Werte annehmen: `selektiert`, `deselektiert` und `unbekannt`. Wird der Status eines Features festgelegt, werden alle automatischen Werte angepasst. Die Anpassung erfolgt mithilfe eines SAT-Solvers. Dieser berechnet, ausgehend von der aussagenlogischen Repräsentation des Feature-Modells und bekannter Selektionsstati die automatischen Selektionsstati. Mit diesen automatischen Selektionsstati ist es möglich zu prüfen, ob bestimmte Konfigurationen möglich sind. Außerdem kann ermittelt werden, ob

[1] http://www.pure-systems.com/pure_variants.49.0.html

durch die Selektion beziehungsweise Deselektion von Features ein Selektionsstatus eines anderen Features impliziert wird.

Ob ein Feature obligatorisch ist, kann ermittelt werden, indem alle manuellen Selektionsstati auf `unbekannt` gesetzt werden. Alle Features, die dann den automatischen Status `selektiert` haben, sind obligatorisch. Wenn ein Feature obligatorisch für eine Methode ist, muss dieses immer automatisch selektiert werden, wenn eines der Features, die die Methode enthalten selektiert ist. Um zu ermitteln, welche Features eine Methode enthalten, sind die *Feature Structure Trees* aus *FeatureHouse* nötig. Ist ein Knoten für die Methode in dem *Feature Structure Tree* eines Features enthalten, enthält das Feature diese Methode.

4.3 Zusammenfassung

Die vorgeschlagenen Regeln für die Einführung neuer Kompositionsverfahren bei der Metaprodukt-Generierung und für die Optimierung der erzeugten Spezifikationen konnte in den Tools *FeatureIDE* und *FeatureHouse* umgesetzt werden. Um die Erzeugung eines Metaproduktes zu ermöglichen wurde ein Interface als Schnittstelle zwischen den beiden Tools definiert. Die Implementierung der Komposition von Spezifikationen erfolgte in *FeatureHouse*. Die für die Optimierungen und für *Plain Contracting* nötigen Informationen zum Feature-Modell werden in *FeatureIDE* ermittelt. Auf diese Informationen kann *FeatureHouse* durch das Interface zugreifen. Mithilfe dieser beiden Tools kann nun ein Metaprodukt mit optimierten Spezifikationen erstellt werden. Bisher ist jedoch noch nicht klar, welchen Einfluss die Optimierungen auf die Verifikation haben.

5. Evaluierung der Optimierungen

In Kapitel 3 wurden Konzepte für die Erstellung und Optimierung von Spezifikationen in einem Metaprodukt vorgestellt. Danach wurde in Kapitel 4 die Umsetzung dieser Konzepte in den Tools *FeatureHouse* und *FeatureIDE* erläutert. Somit wurde gezeigt, dass die Konzepte umsetzbar sind. Bisher ist aber noch nicht klar, ob die Optimierungen Vorteile oder Nachteile erzeugen. Dies wird daher in diesem Kapitel betrachtet.

Nachdem in Abschnitt 5.1 die Software-Produktlinie vorgestellt wird, anhand der die Vorteile untersucht wurden, wird in Abschnitt 5.2 die für die Generierung benötigte Zeit betrachtet. In Abschnitt 5.3 folgt die Untersuchung der Lesbarkeit der Spezifikationen mit und ohne Optimierungen. In Abschnitt 5.4 folgen die Erkenntnisse für den *Theorem Prover KeK*. Die Auswirkungen der Optimierungen auf den *Model Checker Java Pathfinder* werden in Abschnitt 5.5 betrachtet. In Abschnitt 5.6 werden die Ergebnisse noch einmal zusammengefasst. Außerdem werden Faktoren betrachtet, die die Ergebnisse beeinflusst haben könnten.

5.1 Fallstudie BankAccount

Für die Evaluierung der Optimierungen wurde die Software-Produktlinie BankAccount von SPL2go[1] gewählt. Diese hatte auch Meinicke verwendet, sodass eine bessere Vergleichbarkeit erzielt wird. In Abbildung 5.1 auf der nächsten Seite ist das zu der Software-Produktlinie gehörende Feature-Modell abgebildet. Es sind insgesamt acht Features vorhanden.

Das Feature *BankAccount* enthält die Basis-Implementierung. Mit dieser Implementierung können bereits Konten erstellt werden und Beträge auf oder von diesen Konten gebucht werden. Mit dem Feature *DailyLimit* wird eine Obergrenze für Buchungen pro Tag hinzugefügt. Mit *Interest* ist es möglich, Zinsen zu berechnen. Wenn dieses Feature vorhanden ist, kann durch das Feature *InterestEstimation* auch eine Zinsvorausschau erstellt werden. Mit Hilfe des Features *CreditWorthiness* kann

[1] http://spl2go.cs.ovgu.de

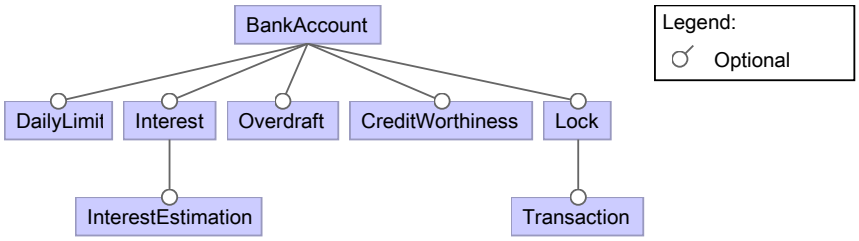

Abbildung 5.1: Feature-Modell der Software-Produktlinie BankAccount

die Kreditwürdigkeit des Kontoinhabers berechnet werden. *Overdraft* führt einen Überziehungskredit ein. Durch das Feature *Lock* kann ein Konto für Änderungen gesperrt werden. Mit Hilfe dieser Sperren und dem Feature *Transaction* sind dann Transaktionen zwischen zwei Konten möglich.

Mit diesen Features lassen sich 72 verschiedene Software-Produkte erstellen. Diese enthalten bis zu drei Klassen mit insgesamt 12 Methoden. Zwei dieser Methoden werden dabei durch ein Feature redefiniert. Eine weitere Methode wird durch zwei Features redefiniert. Alle anderen Methoden sind nur in genau einem Feature vorhanden. Das einzige Kern-Feature dieser Software-Produktlinie ist das Feature *BankAccount*. Durch dieses werden auch alle verfeinerten Methoden eingeführt.

Die Spezifikationen der Methoden in den Feature-Modulen wurden für die Verwendung von *Explicit Contract Refinement* erstellt. So ist ein Vergleich der optimierten Spezifikationen mit den nach Meinickes Konzept [Mei13] erstellten Spezifikationen möglich.

Da es Optimierungen gibt, die bei dieser Software-Produktlinie keine Auswirkungen haben, wurde die Software-Produktlinie um ein zusätzliches Feature für das Loggen von Kontoänderungen eingeführt. Es werden sowohl Änderungen am Kontostand, als auch komplette Transaktionen abgespeichert. Das Feature erweitert die Software-Produktlinie um eine Klasse für Log-Einträge mit vier Funktionen. Außerdem werden die zwei Methoden, die bereits einmal verfeinert wurden, ein weiteres Mal verfeinert. Auch eine Methode, die zuvor nicht verfeinert wurde, wird verfeinert. Diese Methode wird durch das Feature *Transaction* eingeführt.

Bei den verfeinerten Methoden werden die Spezifikationen derart verfeinert, dass immer auch die ursprüngliche Spezifikation erfüllt sein muss. Aus diesem Grund sind keine Implikationen vorhanden, die vereinfacht werden könnten. Um dennoch eine solche Vereinfachung zu erreichen, wurde in der Spezifikation der verfeinernden Methode *update* des Features *DailyLimit* das Schlüsselwort `original` durch die Klausel ersetzt, die bei der Komposition eingefügt werden würde. Damit dies zu einer Vereinfachung der Spezifikation im Metaprodukt führen kann muss zusätzlich ein Bedingung eingefügt werden. Wenn die Bedingung $DailyLimit \Rightarrow Logging$ eingefügt wird, führt dies zu einer Vereinfachung der Spezifikation im Metaprodukt.

Das neue Feature wird in das Feature-Modell als Kind des Wurzel-Features eingefügt. Da in dem Feature eine Methode redefiniert wird, die in *Transaction* eingeführt wird, muss eine zusätzliche Bedingung eingefügt werden. Alternativ könnte das neue Feature auch als Kind des Features *Transaction* eingefügt werden. Aufgrund der zusätz-

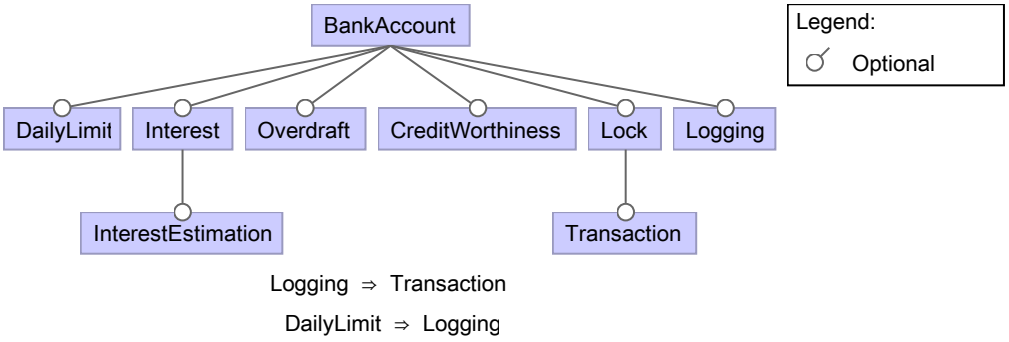

Abbildung 5.2: Feature-Modell der erweiterten Software-Produktlinie BankAccount

lichen Bedingungen reduziert sich die Anzahl der möglichen Software-Produkte auf 60. Das Feature-Modell der erweiterten Software-Produktlinie ist in Abbildung 5.2 abgebildet.

5.2 Zeit zur Generierung des Metaproduktes

Da für die einzelnen Optimierungen Berechnungen anhand des Feature-Modells nötig sind, erfordert dies auch einen Mehraufwand bei der Erzeugung des Metaproduktes. Sollte für die Generierung deutlich mehr Zeit benötigt werden, als bei der Verifizierung eingespart werden kann, ist es möglicherweise günstiger einzelne Optimierungen zu ignorieren.

Zeitmessung bei der Metaprodukt-Generierung

Die Generierung des Metaproduktes erfolgt in mehreren Schritten. Zunächst müssen die Feature-Module komponiert werden. Außerdem ist es nötig, dass die Klasse zur Kodierung der Variabilität erzeugt wird. Diese Klasse ist unabhängig von den Optimierungen immer identisch. Daher kann die für die Erzeugung dieser Klasse nötige Zeit vernachlässigt werden.

Die Zeitmessung erfolgt daher vom Beginn der Komposition der Feature-Module bis zum Abschluss dieser Komposition. Es wurde in den Quelltext vor und nach dem Aufruf der Komposition durch *FeatureHouse* jeweils die Erzeugung eines Zeitstempels eingefügt. Die Differenz dieser Zeitstempel enstpricht dann der, für die Komposition der Feature-Module benötigten Zeit.

Um den Einfluss einzelner Berechnungen zu ermitteln, werden unterschiedliche Optimierungsgrade untersucht. Über boolsche Variablen lassen sich in der Implementierung des Interfaces in *FeatureIDE* einzelne Berechnungen deaktivieren. Die Ausgaben der Funktionen des Interfaces sind dann unabhängig von den Eingabewerten. Die Ausgabe ist analog zu der, die die Implementierung Fallback-Klasse in *FeatureHouse* ausgibt. Die Funktionen des Interfaces wurden gruppiert. Dadurch muss nicht jede Funktion separat betrachtet werden. Außerdem lassen sich einige Funktionen des Interfaces nur nutzen, wenn auch andere Funktionen genutzt werden. Die erste Gruppe umfasst lediglich die Funktion zur Bestimmung, ob ein Feature ein Kern-Feature ist. Im folgenden wird diese Gruppe mit K bezeichnet. Die zweite Gruppe

dient der Bestimmung, ob ein Feature ein Methoden-Kern-Feature ist. Diese wird im folgenden *KM* genannt. Die letzte Gruppe umfasst alle Methoden, zur Ermittlung von Beziehungen zwischen Features und wird mit *FM* abgekürzt. Es wird die Generierungszeit für ein Metaprodukt für alle Kombinationen dieser Gruppen ermittelt und verglichen.

Ergebnisse

In Abbildung 5.3 sind die Ergebnisse dieser Messungen dargestellt. Es wird deutlich, dass die einzelnen Optimierungen unterschiedlichen Einfluss auf die Generierungszeit haben. Der unterschiedliche Einfluss auf die Generierungszeit lässt sich auf den unterschiedlichen Aufwand bei den nötigen Berechnungen zurückführen. Um zu bestimmen, ob ein Feature ein Kernfeature ist, muss lediglich geprüft werden, ob das Feature in einer Konfiguration immer automatisch selektiert ist. Um zu prüfen, ob ein Feature ein Kern-Feature für eine Methode ist, müssen mehrere Konfigurationen geprüft werden. Für jedes Feature, dessen Feature-Modul die Methode enthält, wird eine Konfiguration erzeugt und überprüft. Für die Ermittlung von Beziehungen zwischen den Features sind deutlich mehr Berechnungen erforderlich.

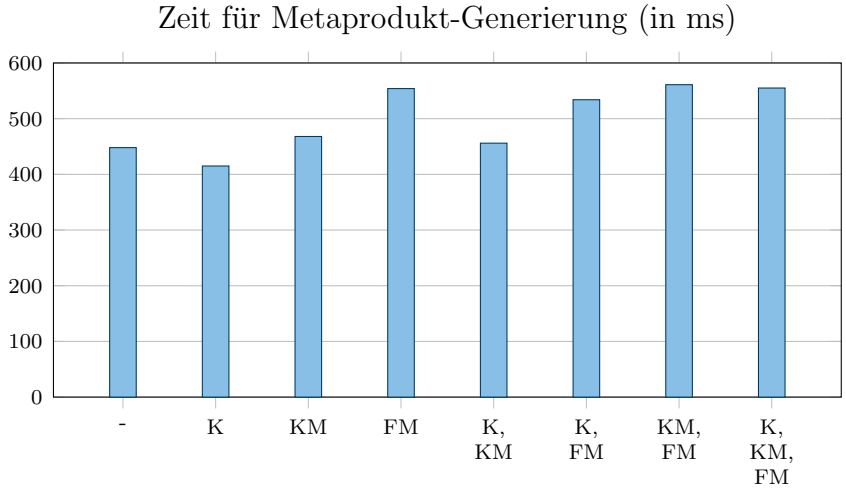

Abbildung 5.3: Generierungszeit des Metaproduktes mit unterschiedlichen Optimierungsgraden

Bei der Kombination der Gruppen wird deutlich, dass der Mehraufwand sich nicht addiert. Stattdessen nimmt dieser teilweise sogar noch geringfügig ab. Eine Grund hierfür könnte sein, dass einige aufwendigere Berechnungen nicht mehr nötig sind, wenn bereits andere Berechnungen durchgeführt wurden. Ein Beispiel hierfür sind Beziehungen zwischen Features. Diese sind nicht mehr relevant, wenn eines der Features ein Methoden-Kern-Feature ist. Der Fall, dass dieses Feature nicht vorhanden ist, muss nicht untersucht werden.

Insgesamt steigt der Aufwand für die Generierung des Metaproduktes der in Abschnitt 5.1 beschriebenen Software-Produktlinie um 107 ms. Bei dieser Software-Produktlinie entspricht dies einer Steigerung um fast 25 %. Die insgesamt benötigte Zeit ist aber dennoch nur ein Bruchteil der Zeit, die für die Verifikation benötigt wird.

5.3 Lesbarkeit der erzeugten Spezifikation

Durch die in Kapitel 3 vorgeschlagenen Optimierungen werden die Spezifikationen im Metaprodukt vereinfacht. Dadurch werden diese für Entwickler leichter lesbar. Das ist dann von Vorteil, wenn die Software-Produktlinie erweitert und dabei auf bereits vorhandene Methoden zurückgegriffen werden soll. Der Entwickler kann dann im Metaprodukt erkennen, welche Spezifikationen eine Methode erfüllt. Auch bei der Verifikation ist eine bessere Lesbarkeit von Vorteil. Wenn Fehler gefunden werden, erleichtert eine bessere Lesbarkeit der Spezifikationen das Nachvollziehen des Fehlers. Wie die Lesbarkeit erhöht wird, wird in diesem Kapitel betrachtet.

Ermittlung der Lesbarkeit

Um die Lesbarkeit von Spezifikationen zu bewerten, wurde sich für zwei Kennzahlen entschieden. Zum einen wird die Anzahl der Klauseln im Metaprodukt betrachtet. Je mehr Klauseln eine Methode enthält, desto komplexer wird die Spezifikation. Allerdings lässt sich die Anzahl der Klauseln auch auf maximal zwei Klauseln pro Methode reduzieren, wenn alle Vor- beziehungsweise Nachbedingungen durch eine Konjunktion zu einer Klausel zusammengefasst werden.

Da durch die Optimierungen aber auch einzelne Klauseln gekürzt wurden, wird zusätzlich die Anzahl der logischen Operationen betrachtet. Es werden dabei nur die bei der Metaprodukt-Generierung eingefügten Operationen betrachtet. Dies sind das logische Und (`&&`), Oder (`||`) und die Implikation (`==>`). Wenn Klauseln durch eine Konjunktion zusammengefasst werden, erhöht sich somit auch diese Kennzahl.

Beide Kennzahlen werden sowohl für das Metaprodukt ermittelt ohne Optimierungen, als auch für das Metaprodukt mit den optimierten Spezifikationen. Da die Spezifikationen in den einzelnen Feature-Modulen diese Kennzahlen beeinflussen, werden diese auch für die Feature-Module bestimmt und die Summe als Vergleichswert ermittelt. Es wird lediglich die unveränderte Software-Produktlinie betrachtet.

Ergebnisse

Die Anzahl der Klauseln in den Spezifikationen wird in Abbildung 5.4 auf der nächsten Seite verglichen. Die roten Säulen repräsentieren die Anzahl der Klauseln im Metaprodukt, das mit der Implementierung von Meinicke, also ohne Optimierungen, erstellt wurde. Die blauen Säulen stellen die Anzahl der Klauseln in den optimierten Spezifikationen nach Kapitel 3 dar und die grünen Säulen repräsentieren als Vergleichswert die Anzahl der Klauseln in den Feature-Modulen.

Es fällt auf, dass die Metaprodukte deutlich mehr requires-Klauseln, also Vorbedingungen enthalten, als in den Feature-Modulen vorhanden sind. Dies resultiert daraus, dass pro Methode bis zu zwei zusätzliche Vorbedingungen eingefügt werden. Außerdem werden im Metaprodukt bei der Redefinition von Methoden zwei Methoden angelegt (siehe Abschnitt 3.1.2). Beide Methoden enthalten Spezifikationen.

Es wird jedoch deutlich, dass die Anzahl der Klauseln in den optimierten Spezifikationen deutlich reduziert ist. Die optimierten Spezifikationen enthalten 28 % weniger requires-Klauseln als die ursprünglich erzeugten Spezifikationen. Werden sowohl Vor-

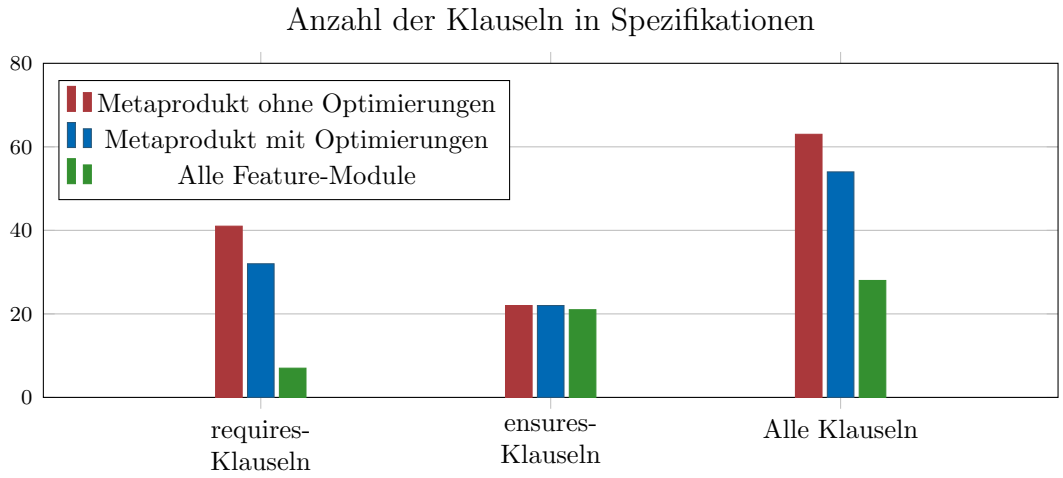

Abbildung 5.4: Anzahl der Klauseln in den Spezifikationen

als auch Nachbedingungen betrachtet, reduziert sich die Anzahl der Klauseln noch um 16 %, da bei den Nachbedingungen weniger Einsparungen erzielt werden konnten. Der Grund für den unterschiedlichen Umfang der Einsparung ist, dass nicht bei jeder Methode die beiden zusätzlichen Vorbedingungen nötig sind.

Auch die Anzahl der logischen Operationen konnte in den optimierten Spezifikationen deutlich reduziert werden. In Abbildung 5.5 wird die Anzahl der logischen Operatoren in den zwei Metaprodukten und in den Feature-Modulen dargestellt.

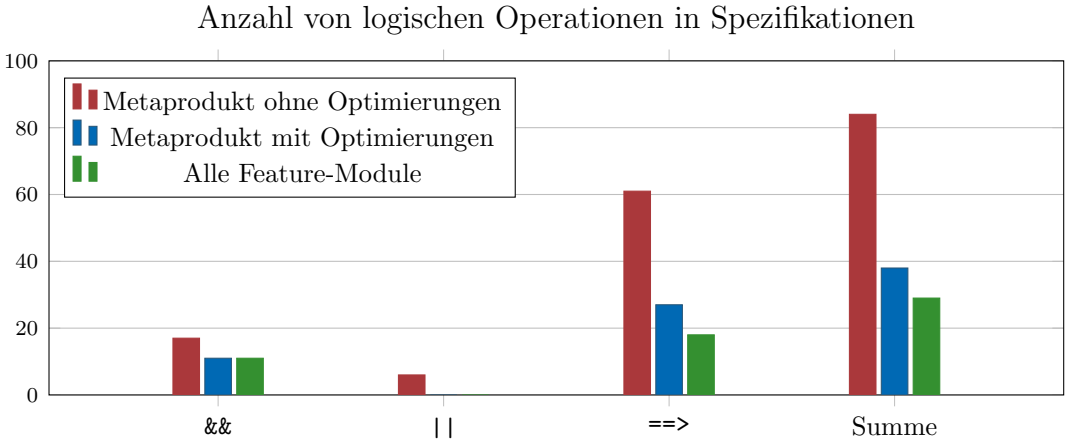

Abbildung 5.5: Anzahl der logischen Operationen in den Spezifikationen

Da die redefinierten Methoden alle im Feature *BankAccount* definiert sind, enthalten die optimierten Spezifikationen keine Disjunktionen der Features mehr. Bei Methoden, die nicht im Feature-Modul des Features *BankAccount* definiert wurden, besteht diese Disjunktion nur aus einem Feature, sodass kein `||` vorhanden ist. Die logischen Unds konnten eingespart werden, da bei den redefinierten Methoden keine Konjunktion mit der ursprünglichen Spezifikation mehr nötig ist. Die größte Einsparung ist jedoch bei den Implikationen erkennbar, da die optimierten Spezifikationen keine Implikation mit dem Selektionsstatus enthalten, wenn bekannt ist, dass das Feature immer vorhanden ist, wenn die Methode aufgerufen wird.

Die Anzahl der logischen Operationen konnte so von 84 auf 38 mehr als halbiert werden. Insgesamt enthält das Metaprodukt mit den optimierten Spezifikationen nur neun logische Operationen mehr, als die Spezifikationen in den Feature-Modulen.

Durch die Optimierungen in den Spezifikationen konnte die Lesbarkeit deutlich gesteigert werden. Die optimierten Spezifikationen enthalten sowohl weniger Klauseln, als auch weniger logische Operationen als die Spezifikationen im Metaprodukt ohne die Optimierungen.

5.4 Verifikation mittels Theorem Proving

Ein weiteres Ziel der Optimierungen in den Spezifikationen im Metaprodukt war es, den Verifikationsaufwand zu reduzieren. Eine Möglichkeit der Verifikation ist mithilfe eines *Theorem Provers* die Korrektheit eines Programms entsprechend der Spezifikationen zu beweisen. Ob die Optimierungen die Effizienz der Beweise erhöhen, wird im folgenden betrachtet.

Aufwandsmessung für Theorem Proving

Bei der Optimierung der Spezifikationen werden verschiedene Regeln angewendet. Um den Einfluss einzelner Regeln zu bestimmen, werden die entsprechenden Optimierungen aktiviert beziehungsweise deaktiviert. Dies erfolgt analog zu der Einteilung zur Messung der Generierungszeit in Abschnitt 5.2.

Für die Verifikation wird der *Theorem Prover MonKey* verwendet. *Monkey* ist ein frei verfügbares Eclipse-Plugin. Dieses Tool gibt für jede Methode an, ob ein Beweis erfolgreich durchgeführt werden konnte. Zusätzlich werden die jeweils nötigen Beweisschritte, Verzweigungen und die erforderliche Zeit pro Methode ausgegeben. Diese Ausgaben werden betrachtet und verglichen. Dabei werden sowohl die unveränderte, als auch die erweiterte Software-Produktlinie untersucht.

Da *MonKey* mit logischen Ausdrücken besser arbeiten kann, als mit Methodenaufrufen, wird zusätzlich analysiert, wie sich die Effizienz ändert, wenn die Vorbedingung `FM.FeatureModel.valid()` durch den logischen Ausdruck ersetzt wird, der innerhalb der Methode geprüft wird. Dadurch wird allerdings die Lesbarkeit wieder herabgesetzt, da die aussagenlogische Darstellung des Feature-Modells sehr komplex werden kann.

Ergebnisse

Die einzelnen Optimierungen haben unterschiedlichen Einfluss auf die benötigte Zeit und die Anzahl der Beweisschritte. Entgegen der Erwartungen führt das Entfernen der Disjunktion aller Features, die eine Methode enthalten dazu, dass die Effizienz des *Theorem Provers* sinkt. In Abbildung 5.6 auf der nächsten Seite sind die gemessenen Zeiten, Beweisschritte und Verzweigungen für die unveränderte Software-Produktlinie mit dem Methodenaufruf von `valid` in der Vorbedingung dargestellt. Jede Säule repräsentiert eine Menge der ausgewählten Optimierungsregeln. *K* steht für die Nutzung des Wissens über Kern-Features, *KM* für die Nutzung des Wissens über Kern-Features der Methode und *FM* für die Nutzung des Wissens über Beziehungen zwischen den Features aus dem Feature-Model.

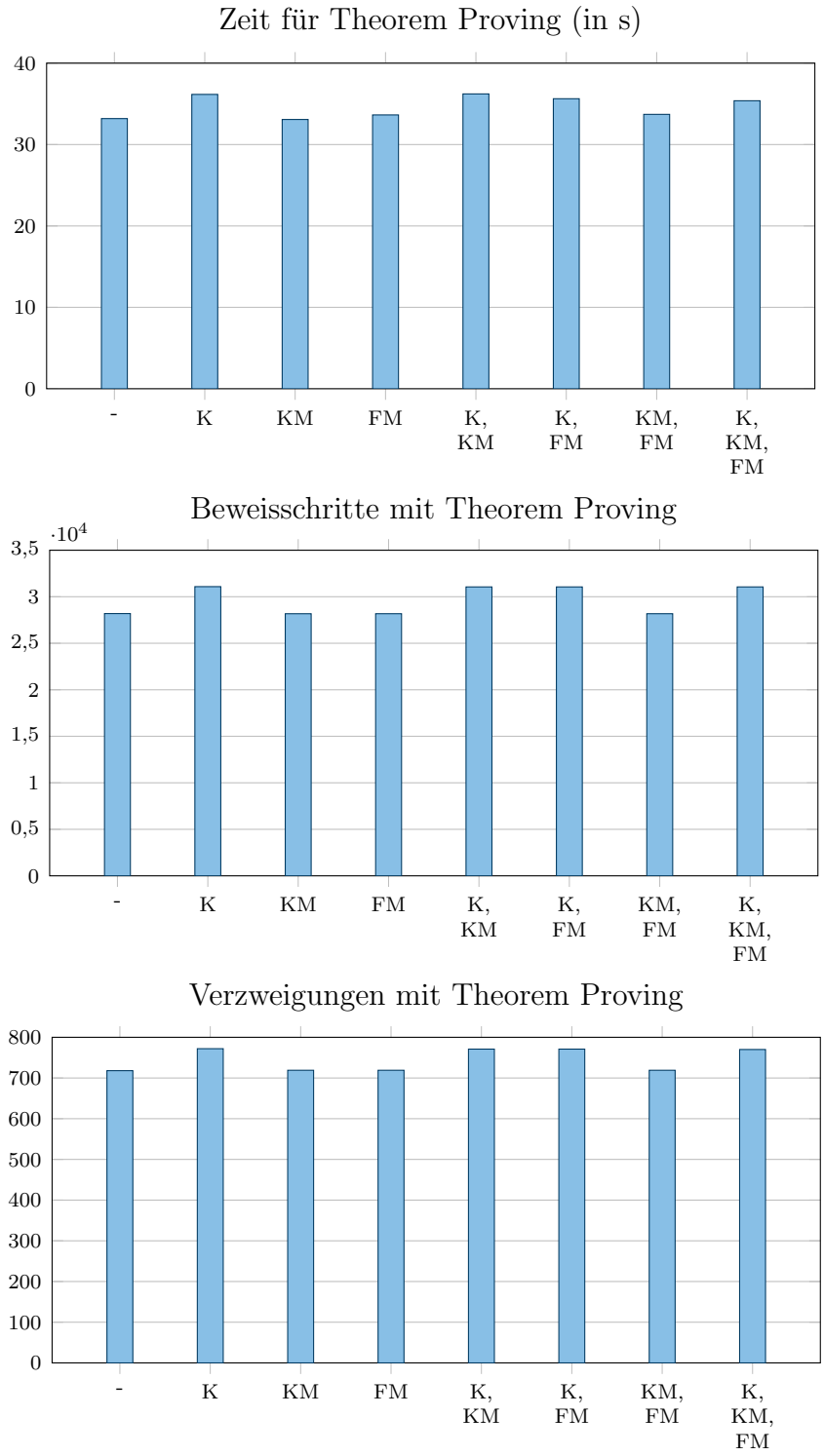

Abbildung 5.6: Benötigte Zeit, Beweisschritte und Verzweigungen bei der Verifikation der unveränderten Software-Produktlinie mit *MonKey*

Die Kenntnis über die Kernfeatures des Feature-Modells führt dazu, dass die Disjunktionen der Features, deren Feature-Module die spezifizierte Methode enthalten, entfernt werden können. Wird dieses Wissen genutzt, führt dies dazu, dass *MonKey* mehr Beweisschritte und somit auch mehr Zeit benötigt, um die Korrektheit der Software-Produktlinie zu beweisen. Die Verwendung der anderen Optimierungsregeln führt jeweils dazu, dass die Anzahl der Beweisschritte zurück geht. Allerdings beträgt der Unterschied unter 0,1 %, sodass diese bei der unveränderten Software-Produktlinie kaum Vorteile bringt. Die benötigte Zeit für den Beweis steigt sogar an. Zeitmessungen müssen aber kritisch betrachtet werden, da diese leicht durch andere Prozesse beeinflusst werden können.

Wird die Vorbedingung `FM.FeatureModel.valid()` allerdings durch die aussagenlogische Repräsentation des Feature-Modells ersetzt, ändern sich diese Erkenntnisse. Bereits ohne Optimierungen reduziert sich die Anzahl der Beweisschritte dadurch von 28190 mit der Vorbedingung `FM.FeatureModel.valid()` auf 8976 mit der aussagenlogischen Repräsentation des Feature-Modells. Das entspricht einer Reduzierung um 68 %. Abbildung 5.7 stellt die so benötigten Beweisschritte und Verzweigungen nach dem Ersetzen der Vorbedingung dar. Die Unterschiede zwischen den Anzahlen der Beweisschritte bei unterschiedlichen Optimierungen sind sehr gering. Die Differenz zwischen der niedrigsten (8943) und der höchsten Anzahl (8976) beträgt lediglich 33 Beweisschritte. Das entspricht etwa 0,4 %. Dennoch sind Unterschiede zu der Verifikation mit der Vorbedingung `FM.FeatureModel.valid()` erkennbar. Ähnlich verhalten sich die Verzweigungen. Der Unterschied zwischen dem niedrigsten (196) und dem höchsten Wert (197) ist lediglich eine Verzweigung. Um diese Unterschiede in Abbildung 5.7 sichtbar zu machen, beginnen die Skalen nicht bei 0.

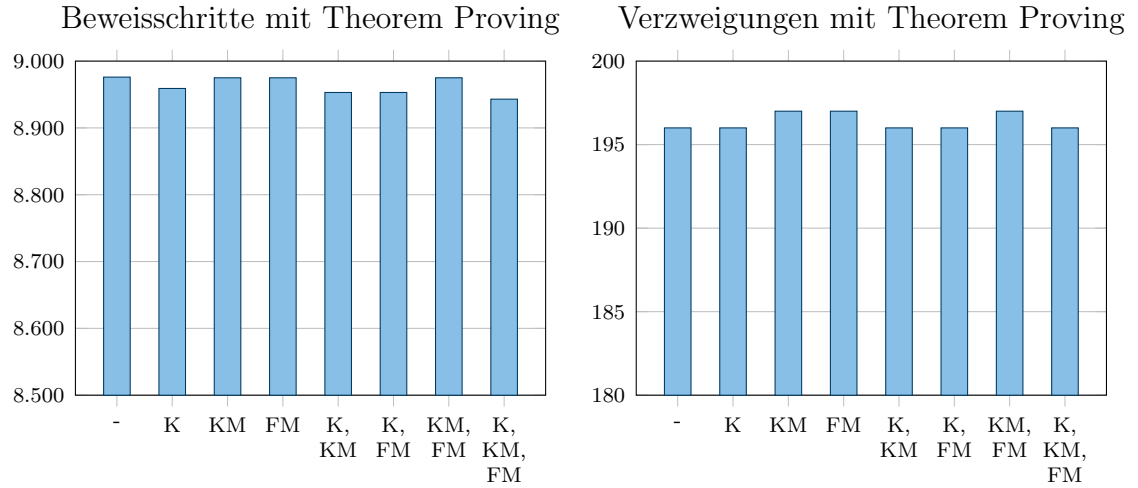

Abbildung 5.7: Benötigte Beweisschritte und Verzweigungen bei der Verifikation der unveränderten Software-Produktlinie mit *MonKey* mit Ersetzung der Vorbedingung `FM.FeatureModel.valid`

Der Einfluss von Kenntnissen über Methoden-Kern-Features und Beziehungen zwischen Features bleibt bei der unveränderten Software-Produktlinie sehr gering. Allerdings wird die Anzahl der Beweisschritte durch Kenntnis über Kern-Features reduziert. Wird dieses Wissen mit dem Wissen über Methoden-Kern-Features und

Beziehungen zwischen Features kombiniert, ist die Einsparung sogar noch größer, auch wenn diese Kenntnisse einzeln einen sehr geringen Einfluss haben.

Bei der unveränderten Software-Produktlinie werden durch Methoden-Kern-Features und Beziehungen zwischen Features kaum Vorteile erreicht. Dies liegt vor allem daran, dass in der Software-Produktlinie kaum Verknüpfungen zwischen den einzelnen Features vorhanden sind, und nur drei Methoden redefiniert werden, die alle im Feature-Modul des einzigen Kern-Features der Software-Produktlinie enthalten sind. Weiterhin enthalten alle redefinierenden Spezifikationen das Schlüsselwort *original* in den Vor- und Nachbedingungen. Dadurch entstehen keine Feature-Kombinationen mehr, die vereinfacht oder gelöscht werden könnten.

In der erweiterten Software-Produktlinie wurde dies jedoch provoziert. Daher haben die Optimierungsgrade auch einen größeren Einfluss auf die Verifikation mithilfe von *MonKey*. In Abbildung 5.8 auf der nächsten Seite sind die ermittelten Beweisschritte, Verzweigungen und die nötige Zeit für die Verifikation der erweiterten Software-Produktlinie dargestellt. In den Spezifikationen wurde dabei die aussagenlogische Repräsentation des Feature-Modells anstelle der Methode `valid` genutzt.

Das Nutzen des Wissens über Kern-Features und Beziehungen zwischen Features alleine hat dabei keinen beziehungsweise nur einen geringen Einfluss auf die Verifikation. Die Anzahl der Beweisschritte bleibt annähernd identisch. Die Zahl der Verzweigungen nimmt leicht ab. Die benötigte Zeit nimmt bei der Verwendung von Kern-Features zur Optimierung sogar zu.

Wenn das Wissen über Methoden-Kern-Features genutzt wird, steigt die Effizienz der Verifikation stärker an. Es sind 2 % weniger Beweisschritte nötig, als ohne die Verwendung der Optimierungen. Die Anzahl der Verzweigungen im Beweis sinkt um 1 % und die Zeit um 5 %.

Verglichen mit der prozentualen Steigerung der Generierungszeit scheint dies sehr gering zu sein. Allerdings ist die absolute Einsparung deutlich größer. Während die Generierungszeit bei der Optimierung der Spezifikationen mithilfe von Kern-Features für Methode lediglich 20 ms länger war, als ohne die Optimierungen, beträgt die Einsparung bei der Verifikation 7,5 s.

Wird das Wissen über Kern-Features der Methode mit weiteren Optimierungen kombiniert, steigt der Aufwand für die Verifikation wieder. Scheinbar ist der Beweis für den *Theorem Prover* leichter zu schließen, wenn die Spezifikationen zusätzliche Klauseln enthalten. Welche Optimierungen von Vorteil sind, scheint auch von der Software-Produktlinie und dem zugrunde liegenden Feature-Modell abzuhängen. Bei der unveränderten Software-Produktlinie hatte das Wissen über Kern-Features für Methoden keine Vorteile gebracht.

Beim *Theorem Prover MonKey* bringen die vorgeschlagenen Optimierungen unterschiedlich starke Vorteile. Zum Teil sind die Optimierungen sogar von Nachteil für die Effizienz der Verifikation. Wird allerdings in den Vorbedingungen der Methodenaufruf `FM.FeatureModel.valid()` durch die aussagenlogische Repräsentation des Feature-Modells ersetzt, erhöht sich die Effizienz durch die Verwendung der Optimierungen stärker. Außerdem wird dadurch die Effizienz des Tools auch ohne Optimierungen deutlich gesteigert.

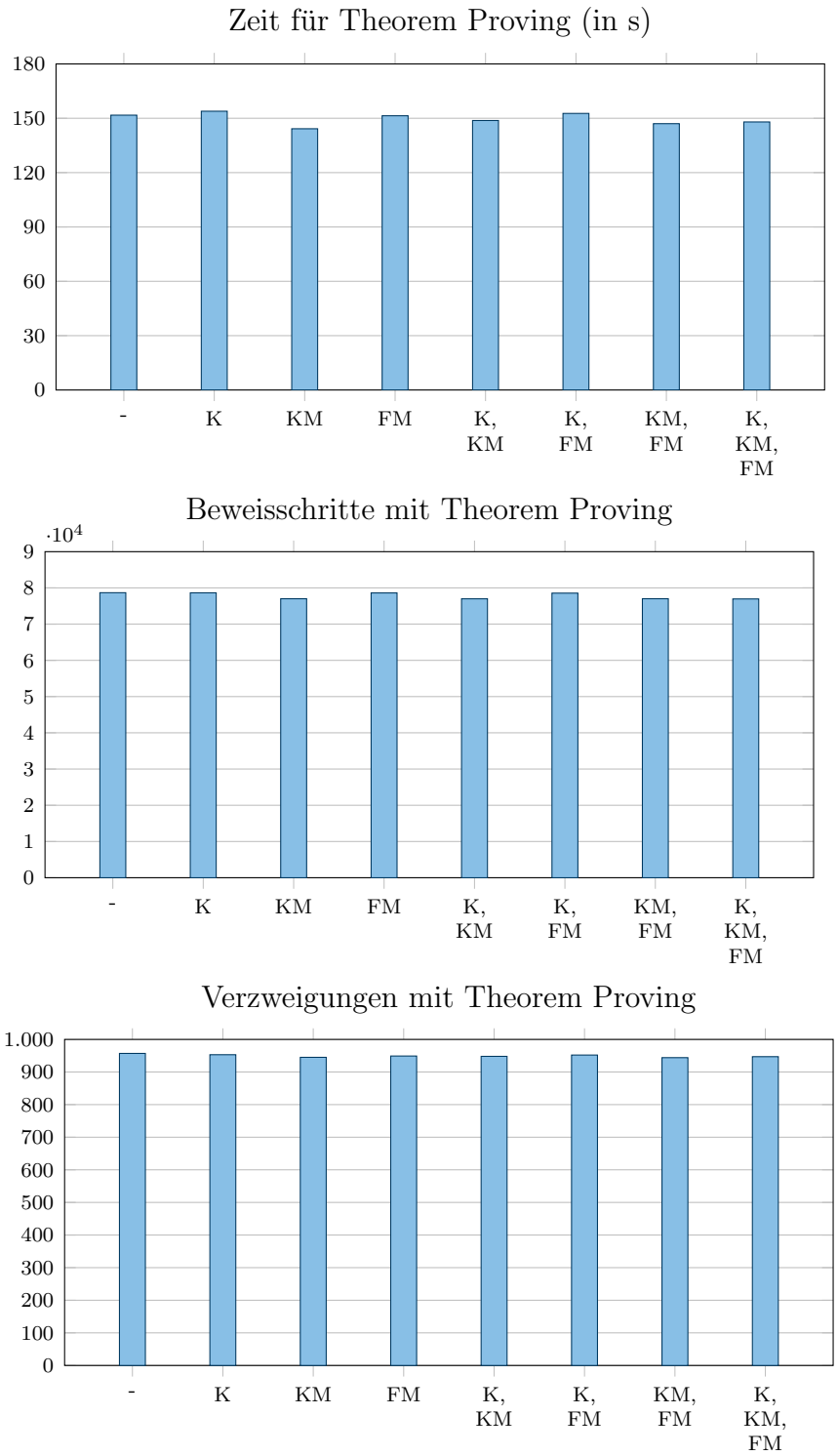

Abbildung 5.8: Benötigte Beweisschritte, Verzweigungen und Zeit bei der Verifikation der erweiterten Software-Produktlinie mit *MonKey* mit Ersetzung der Vorbedingung `FM.FeatureModel.valid()`

5.5 Verifikation mittels Model Checking

Eine andere Möglichkeit der Verifikation ist *Model Checking*. Mithilfe der Tools *OpenJML* und *Java Pathfinder* können die Spezifikationen im Metaprodukt genutzt werden, um *Model Checking* durchzuführen. Auch hierbei können die Optimierungen von Vorteil sein.

Aufwandsmessung für Model Checking

Der Model Checker *Java Pathfinder* ermittelt von einem Startzustand ausgehend alle erreichbaren Zustände. Wenn dabei ein unerlaubter Zustand erreicht wird, enthält das zu verifizierende Programm Fehler. Wird kein unerlaubter Zustand erreicht bedeutet dies jedoch nicht, dass das Programm fehlerfrei ist.

Das Tool *Java Pathfinder* kann jedoch die JML-Spezifikationen nicht direkt verwenden. Daher werden diese mithilfe des Tools *OpenJML* in *Runtime-Assertions* umgewandelt. Analog zum *Theorem Proving* wird wiederum zwischen verschiedenen Optimierungsgraden unterschieden, um den Einfluss einzelner Optimierungen zu erkennen. Der *Java Pathfinder* gibt am Ende der Berechnungen aus, wie viel Zeit benötigt wurde, wie viele Zustände besucht wurden und wie viele Anweisungen ausgeführt wurden. Die so erzielten Werte der einzelnen Optimierungsgrade lassen sich dann vergleichen.

Auch bei der Verifikation mittels *Model Checking* werden sowohl die Auswirkungen der Optimierungen auf die Verifikation unveränderten Software-Produktlinie, als auch der erweiterten Software-Produktlinie betrachtet. Auf diese Weise wird zum einen der Einfluss von Erweiterungen auf die Verifikation deutlich. Zum anderen unterscheiden sich die Software-Produktlinien in der Anzahl der möglichen Optimierungen.

Ergebnisse

Die für die Verifikation der unveränderten Software-Produktlinie ermittelten Zeiten und die Anzahl der Anweisungen sind in Abbildung 5.9 auf der nächsten Seite dargestellt. Die Anzahl der Zustände ist bei allen Optimierungen identisch. Die Optimierungen habe keinen Einfluss auf die Anzahl der Zustände. Dies lässt sich dadurch erklären, dass die Spezifikationen nicht zu einer Zustandsänderung führen. Um eine Zustandsänderung herbeizuführen müsste der Quellcode der Methoden geändert werden. Es werden aber lediglich *Assertions* eingefügt, die keinen Einfluss auf den Zustand haben, sondern signalisieren, wenn ein Zustand ungültig ist.

Alle Optimierungen haben einen positiven Effekt auf die Effizienz der Verifikation der Software-Produktlinie. Sowohl die benötigte Zeit, als auch die Anzahl der nötigen Anweisungen sinkt. Den größten Vorteil liefert bei der unveränderten Software-Produktlinie das Nutzen der Informationen über Kern-Features. Dies liegt daran, dass viele Methoden im Kern-Feature *BankAccount* definiert wurden und somit in jedem Software-Produkt enthalten sind. Bei diesen Methoden entfällt daher die Disjunktion aller Features, die diese Methode enthalten. Dadurch fallen viele zu überprüfende *Assertions* weg.

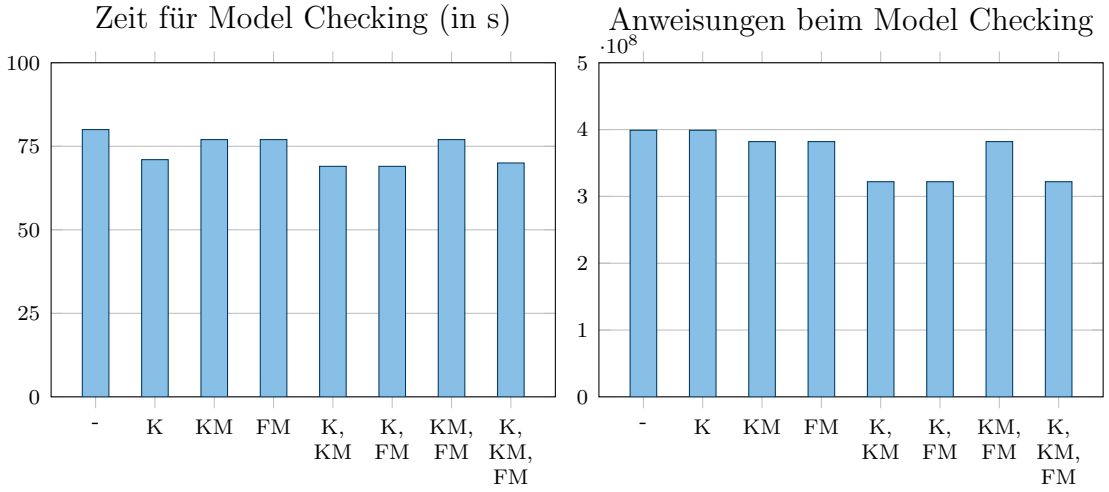

Abbildung 5.9: Benötigte Zeit und Anzahl der Anweisungen bei der Verifikation der unveränderten Software-Produktlinie mit *Java Pathfinder*

Methoden-Kern-Features und Beziehungen zwischen Features haben bei dieser Software-Produktlinie den gleichen Einfluss auf die Effizienz der Verifikation. In beiden Fällen werden die gleichen Klauseln in den Spezifikationen gestrichen. Die Vereinfachungen, die innerhalb einer Klausel vorgenommen werden können, steigern die Effizienz des *Java Pathfinder* scheinbar nicht. Aus diesem Grund sind auch die Zeiten und die Anzahl der Anweisungen jeweils bei der Kombination mit den Vereinfachungen aufgrund des Wissens über Kern-Features identisch. Ebenfalls identisch dazu sind die Werte bei der Kombination aller Optimierungen. Die Zeitersparnis bei der Verifikation beträgt dann 10 s.

Bei der Verifikation der erweiterten Software-Produktlinie werden leicht veränderte Werte für die benötigte Zeit und die Anzahl der Anweisungen gemessen. In Abbildung 5.10 auf der nächsten Seite werden diese dargestellt. Aufgrund des hinzugefügten Features steigt die benötigte Zeit und die Anzahl der Anweisungen. Insgesamt bleiben die Ergebnisse aber ähnlich.

Den größten Vorteil bietet auch in diesem Fall wieder das Wissen über Kern-Features. Allerdings werden bei der Optimierung mittels Beziehungen zwischen Features mehr Klauseln entfernt, als durch die Optimierung mittels Kern-Features für Methoden. Aus diesem Grund ist auch die Anzahl der Anweisungen und somit die benötigte Zeit bei der Verwendung von Beziehungen zwischen Features geringer.

Bei der unveränderten Software-Produktlinie war die Anzahl der Instruktionen bei der Kombination aller Optimierungen noch identisch zu der Anzahl der Anweisungen bei der Kombination des Wissens über Kern-Features mit einer der beiden anderen Optimierungen. Bei der erweiterten Software-Produktlinie ist dies nicht mehr der Fall. Hier ist die Kombination aller Optimierungen am günstigsten für die Verifikation der Software-Produktlinie. Die eingesparte Zeit bei der Verifikation beträgt bei der Anwendung aller Optimierungen 22 s.

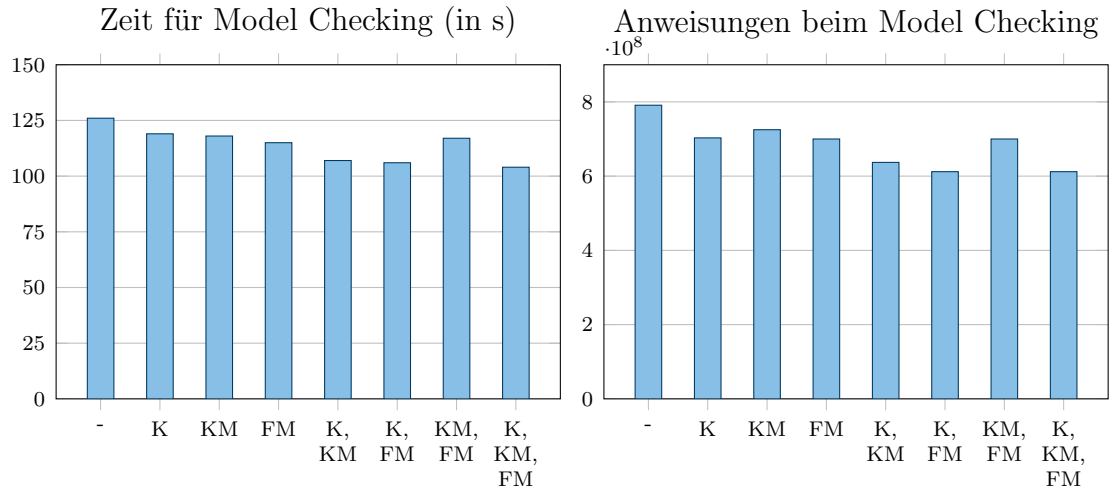

Abbildung 5.10: Benötigte Zeit und Anzahl der Anweisungen in Mio. bei der Verifikation der erweiterten Software-Produktlinie mit *Java Pathfinder*

5.6 Zusammenfassung

Die in in Kapitel 3 vorgestellten Regeln zur Generierung und Vereinfachung von Spezifikationen in einem Metaprodukt wurden umgesetzt. Es wurden so Metaprodukte für zwei Software-Produktlinien erzeugt. Dabei wurden einzelne Regeln für die Vereinfachung der Spezifikationen ignoriert, um die Auswirkung dieser Optimierungen auf die Effektivität der Verifikation zu ermitteln.

Zunächst wurde daher die benötigte Zeit für die Generierung des Metaproduktes betrachtet. Aufgrund der zusätzlichen Berechnungen, um die Optimierungen auszuführen steigt die für die Generierung benötigte Zeit an. Der relative Anstieg der Generierungszeit ist mit etwa 25% zwar recht hoch, die benötigte Zeit für die Generierung beträgt dennoch nur einen Bruchteil der, für die Verifikation benötigten Zeit.

Durch die Optimierungen konnte die Lesbarkeit der erzeugten Spezifikationen deutlich erhöht werden. Gegenüber den Spezifikationen, die mit der Implementierung von Meinicke [Mei13] erzeugt wurden, konnte sowohl die Anzahl der Klauseln reduziert werden, als auch die Anzahl der logischen Operationen innerhalb der Spezifikationen.

Die Einsparungen durch die Optimierungen bei der Verifikation sind sowohl von der Software-Produktlinie, als auch vom verwendeten Verifikationsverfahren abhängig. Beim *Theorem Proving* mit *MonKey* ist der Nutzen der Optimierungen sehr von der Software-Produktlinie abhängig. Während bei der unveränderten Software-Produktlinie die benötigte Zeit ansteigt, wenn die Spezifikationen vereinfacht werden, führen einige Optimierungen bei der Spezifikation des Metaproduktes der erweiterten Software-Produktlinie zu Einsparungen bei der Verifikation. Der Nutzen der Optimierungen scheint stärker zu sein, je mehr Beziehungen zwischen den Features existieren und je mehr Redefinitionen von Methoden enthalten sind. Weiterhin wurde festgestellt, dass die Verifikation der Software-Produktlinie wesentlich effizienter ist, wenn die Spezifikationen statt eines Methodenaufrufs zur Überprüfung der Gültigkeit der Konfiguration der dort enthaltene aussagenlogische Ausdruck direkt in die Spezifikation eingefügt wird.

Bei der Verifikation mit dem *Model Checker Java Pathfinder* haben die Optimierungen stärkere Auswirkungen, als beim *Theorem Prover*. Allerdings sind die Optimierungen nur dann von Vorteil, wenn dadurch Klauseln aus den Spezifikationen entfernt werden. Die Einsparungen bei der Verifikationszeit sind jedoch in jedem Fall größer, als die benötigte Zeit für die Generierung des Metaproduktes.

Die gemessenen Zeiten müssen jedoch auch kritisch betrachtet werden. Im Gegensatz zu den anderen ermittelten Kennzahlen sind diese nicht deterministisch. Die Zeitmessung kann durch andere Prozesse, die auf dem Computer laufen beeinflusst werden. Um diese Einflüsse so gering wie möglich zu halten, wurden alle Zeiten mehrmals gemessen und dann der Durchschnitt der ermittelten Zeiten bestimmt. Weiterhin werden die Zeiten durch Caching beeinflusst. Dies wird dadurch deutlich, dass bei einer erneuten Messung die benötigte Zeit deutlich reduziert ist. Daher wurden die Zeiten für die ersten Berechnungen ignoriert.

6. Verwandte Arbeiten

Andere Arbeiten haben sich bereits mit ähnlichen oder verwandten Themen befasst. Diese werden in diesem Kapitel vorgestellt.

Design By Contract

Desgin By Contract wurde zunächst für Objekt-orientierte Programmierung vorgeschlagen. Meyer [Mey92] beschreibt die Anwendung für Eiffel. Leavens und Cheon [LC03] geben eine Einführung für die Anwendung von Design by Contract in Java-Programmen mithilfe von JML. Burdy et al. [BCC+03] stellen einige Tools vor, die JML unterstützen.

Thüm et al. [TSK+12] haben vorgeschlagen Desgin By Contract auch für feature-orientierte Programmierung zu verwenden. Sie beschreiben auch einige Verfahren zur Komposition von Kontrakten bei der Erstellung eines Software-Produktes. Diese und weitere werden von Benduhn [Ben12] formal beschrieben. Die beschriebenen Kompositionsverfahren wurden für die komplette Software-Produktlinie angewendet. Weigelt [Wei13] beschreibt, dass dieses Vorgehen nicht immer ausreichend ist und schlägt statt dessen vor, das anzuwendende Verfahren je Methode festzulegen. Außerdem gibt er Regeln an, nach denen das Ändern des Kompositionsverfahrens während der Komposition erfolgen kann.

Verifikation

Die Verifikation von Software-Produktlinien kann auf verschiedene Arten erfolgen. Thüm et. al [TAK+14] beschreiben die Produkt-, Feature- und Familien-basierte Analysen, sowie Kombinationen daraus.

Da die Menge an möglichen Produkten bis zu exponentiell in Anzahl der Features steigt, ist die Verifikation jedes einzelnen Produktes häufig zu aufwendig. Daher wird versucht, bei der Produkt-basierten Analyse die Anzahl der analysierten Produkte zu reduzieren und dabei trotzdem noch viele Fehler zu finden. Tartler et al. [TLD+11] beschreiben mit *Configuration Coverage* eine Möglichkeit, um mit einer

möglichst kleinen Menge an Software-Produkten möglichst alle Fehler zu ermitteln. Eine weitere Möglichkeit bietet *Pair-wise* [OMR10] beziehungsweise allgemeiner *t-wise Testing* [PSK+10]. Im Gegensatz zur Verwendung des Metaproduktes für die Verifikation können dabei aber auch Fehler übersehen werden.

Um für die Verifikation nicht alle Software-Produkte generieren und separat zu verifizieren schlagen Post und Sinz [PS08] *Configuration Lifting* vor. Dabei wird die Variabilität, die durch die Konfiguration erreicht wird in die Laufzeit überführt. Auf diese Weise kann Familien-basiert verifiziert werden. Thüm et al. [TSHA12] greifen diesen Ansatz ebenfalls auf. Sie schlagen zusätzlich vor, auch Spezifikationen zu komponieren. Meinicke [Mei13] nutzt und implementiert diesen Ansatz, um Software-Produktlinien mithilfe eines *Theorem Provers* und eines *Model Checkers* zu verifizieren. Diese Arbeit baut auf der Implementierung durch Meinicke [Mei13] auf.

Für bestimmte Analyse-Strategien gibt es auch Ansätze speziell für Software-Produktlinien. Solche Ansätze existieren zum Beispiel für *Type-Checking*. Für *Type-Checking* ist das Metaprodukt weniger geeignet, da immer alle Felder vorhanden sind. Der Aufruf nicht existierender Methoden könnte allerdings durch die zusätzlichen Vorbedingungen ermittelt werden. Kästner und Apel [KA08] beschreiben jedoch einen formalen Ansatz, um zu garantieren, dass alle Software-Produkte korrekt typisiert sind. Apel et al. [AKGL10] haben ein Type-Checker entwickelt, mit dem es möglich ist, sicherzustellen, dass alle gültigen Software-Produkte wohl typisiert sind. In dieser Arbeit wird davon ausgegangen, dass die Software-Produktlinie typsicher ist. Daher sollte bei der Verifikation auch zusätzlich ein *Type-Checker* eingesetzt werden.

Auch für das Theorem Proving gibt es weitere Ansätze, um eine Software-Produktlinie zu verifizieren. Für Software-Produktlinien, die mittels Delta-orientierter Programmierung entwickelt wurden, haben Bruns et al. *Delta-Oriented Slicing* vorgeschlagen. Dabei wird die Korrektheit eines Produktes bewiesen. Die Ergebnisse werden dann als Teilbeweise für den Beweis aller anderen Produkte genutzt. Für Feature-orientierte Programmierung schlagen Thüm et al. [TSKA11] Beweiskomposition vor. Dabei werden für alle Features interaktive Beweisskripte erstellt. Ein Software-Produkt kann dann verifiziert werden, in dem die Beweisskripte der einzelnen Features komponiert und überprüft werden.

7. Zusammenfassung

Die Verifikation ist ein wichtiger Bestandteil des Software-Entwicklungsprozesses. Gerade bei kritischen Sicherheitssystemen ist die Zuverlässigkeit wichtig. Mithilfe von Spezifikationen ist es möglich, die Korrektheit von Programmen zu erhöhen oder sogar zu beweisen. Eine Möglichkeit solche Spezifikationen zu definieren ist *Design By Contract*. Um *Design By Contract* in Java-Programmen nutzen zu können, kann JML verwendet werden.

Software-Produktlinien ermöglichen es, individuelle Software-Produkte mit einer gemeinsamen Code-Basis zusammen zu entwickeln. Dadurch können die Entwicklungskosten stark reduziert werden. Weiterhin reduziert sich auch die Anzahl der möglichen Fehler über alle Software-Produkte, da diese auf der gleichen Code-Basis aufbauen. Durch Interaktionen zwischen einzelnen Feature-Modulen können aber auch neue Fehler entstehen.

Einzelne Produkte der Software-Produktlinie lassen sich in Isolation verifizieren. Bei n Features existieren aber 2^n mögliche Kombinationen aus diesen Features. Alle diese Produkte zu verifizieren kann zu aufwendig sein. Daher wurden verschiedene Verfahren vorgeschlagen, um Mengen von Produkten zu verifizieren, wie zum Beispiel *t-wise Verification* oder *Code Coverage*. Allerdings werden mit diesen Verfahren nicht alle möglichen Produkte verifiziert.

Eine bessere Möglichkeit, bietet die Familien-basierte Verifikation der kompletten Software-Produktlinie. Um mit der möglichen exponentiellen Anzahl an Produkten umzugehen, kann ein Metaprodukt generiert werden. Dabei wird die Variabilität von der Kompilierungszeit in die Laufzeit verschoben. Dieses Metaprodukt kann so alle Software-Produkte simulieren.

Die vorhandenen Regeln zur Generierung des Metaproduktes wurden um Regeln für zusätzliche Kompositionsverfahren bei der Kontrakt-Komposition erweitert. Zusätzlich wurden Regeln definiert, nach denen die erzeugten Spezifikationen vereinfacht werden können. Die Implementierung in den Tools *FeatureIDE* und *FeatureHouse* wurde um diese Regeln erweitert. Durch die, für die Optimierungen nötigen Berechnungen steigt die Zeit, die benötigt wird, um das Metaprodukt zu generieren.

Es wurde festgestellt, dass die Lesbarkeit der Spezifikationen deutlich zunimmt. Bei dem verwendeten *Theorem Prover MonKeY* sind die Optimierungen nur bedingt von Vorteil. Bei der Verifikation mit dem *Model Checker Java Pathfinder* wird die Verifikation deutlich beschleunigt.

Die Lesbarkeit ist vor allem dann interessant, wenn das Metaprodukt genutzt wird, um die Entwicklung beziehungsweise Weiterentwicklung zu erleichtern. Es ist erkennbar, welche Methoden bei welchen Feature-Kombinationen vorhanden sind. Weiterhin kann der Entwickler erkennen, welche Spezifikationen eine Methode bei einer bestimmten Konfiguration erfüllt. Durch die Vereinfachungen wird sowohl die Anzahl der Klauseln, als auch die Anzahl der logischen Operationen innerhalb der Klauseln deutlich reduziert.

Bei der Verifikation mithilfe des *Theorem Proovers MonKey* ist der Nutzen der Optimierungen gering. Teilweise wird durch die Optimierungen der Verifikationsaufwand sogar erhöht. Der Einfluss der Optimierungen ist dabei auch von der Software-Produktlinie abhängig. Bei wenigen Abhängigkeiten zwischen den Features führen die Optimierungen zu einem Mehraufwand bei der Verifikation. Es wurde jedoch auch deutlich, dass der Verifikationsaufwand sehr stark reduziert werden kann, wenn zur Überprüfung der Validität einer Konfiguration statt eines Methodenaufrufs ein entsprechender aussagenlogischer Ausdruck verwendet wird. Wird der aussagenlogische Ausdruck verwendet, reduziert sich der Verifikationsaufwand deutlich.

Wird der *Model Checker Java Pathfinder* für die Verifikation verwendet, reduziert sich der Verifikationsaufwand durch die Optimierungen bei den gewählten Software-Produktlinien. Die Vereinfachungen haben dabei aber nur einen Einfluss, wenn sie dazu führen, dass komplette Klauseln entfernt werden.

Die vorgeschlagenen Optimierungen sind nützlich für die Verifikation von Software-Produktlinien. Die für die Verifikation benötigte Zeit kann dadurch reduziert werden. Die Einsparungen sind größer als der Mehraufwand für die Generierung des Metaproduktes. Die Optimierungen können daher die Verifikation der Software-Produktlinie mithilfe des Metaproduktes beschleunigen.

8. Zukünftige Arbeiten

In dieser Arbeit wurden Optimierungen in Spezifikationen im Metaprodukt einer Software-Produktlinie vorgeschlagen. Diese sind hauptsächlich aufgrund von Beziehungen zwischen Features im Feature-Modell möglich. Es ist auch denkbar, dass durch diese Beziehungen ebenfalls Vereinfachungen innerhalb des generierten Codes innerhalb der Methoden möglich ist.

Es wurde untersucht, wie sich die Optimierungen auf die Verifikation mithilfe des *Theorem Proovers MonKey* und des *Model Checkers Java Pathfinder* auswirken. Es sind aber auch andere Verifikationsmethoden denkbar. Es könnten beispielsweise statische Analysen durchgeführt werden. Auch auf diese könnten die Vereinfachungen Auswirkungen haben.

Die verwendete Software-Produktlinie eines Bank-Kontos und die Erweiterung sind vergleichsweise kleine Software-Produktlinien. Auf größere Software-Produktlinien könnten sich die Vereinfachungen anders auswirken. Es sollte beispielsweise untersucht werden, wie sich die Generierungszeit bei komplexeren Feature-Modellen verhält.

Bei der Verifikation mittels *MonKeY* wurde *Method Inlining* genutzt. Das bedeutet, dass bei aufgerufenen Methoden deren Quellcode ebenfalls analysiert wird. Es besteht aber auch die Möglichkeit statt des Quellcodes die Spezifikationen der aufgerufenen Methode zu verwenden. Gerade wenn die aufgerufenen Methoden bereits bewiesen wurden, kann so der Aufwand reduziert werden. Das Problem bei der Verwendung von Kontrakten ist, dass sich die Beweise häufig nicht schließen lassen.

Beim *Model Checking* kann durch die Zusammenfassung von Zuständen die Effizienz der Verifikation gesteigert werden. Gerade bei der Verifikation des Metaproduktes entstehen sehr viele ähnliche Zustände, die sich nur durch die gewählten Features unterscheiden. Diese können von dem gewählten Tool *Java Pathfinder* nicht zusammengefasst werden, da sich die Werte der Variablen für die Selektionsstati unterscheiden. Erweiterungen des Tools, wie *jpf-bdd* [KvRE+12, vRAR11] könnten dieses Problem lösen. Es müsste untersucht werden, ob diese Erweiterung für das Metaprodukt anwendbar ist.

Weiterhin könnten andere Verifikationsverfahren untersucht werden. In dieser Arbeit wurden *Model Checking* mit *Runtime-Assertions* und *Theorem Proving* betrachtet. Ebenso interessant ist, ob statische Analysen auf das Metaprodukt anwendbar sind. Auch hier wären die Auswirkungen der Optimierungen interessant. Weiterhin könnten Tools für die Generierung von Testfällen untersucht werden. Es müsste geprüft werden, ob diese Tools mit dem Metaprodukt arbeiten können und welche Testfälle diese erzeugen. Unterscheiden sich die Testfälle bei der Anwendung verschiedener Optimierungen in den Spezifikationen? Wie stark unterscheiden sich die erzeugten Testfälle von den Testfällen, die aus den einzelnen Produkten erzeugt werden? Wird die gleiche Testabdeckung erreicht?

In dieser Arbeit wurden nicht alle JML-Konstrukte betrachtet. So ermöglicht JML auch, Spezifikationen für verschiedene Sichtbarkeiten zu definieren [LC03]. Weiterhin ist es möglich, mithilfe des Schlüsselwortes `assignable` anzugeben, welche Variablenwerte verändert werden. Nicht angegebene Variablen dürfen innerhalb der Methode nicht verändert werden. Zunächst müsste die Komposition von Spezifikationen allgemein erweitert werden. Dann muss überprüft werden ob und wie diese Konzepte in das Metaprodukt überführt werden können.

Mithilfe des Metaproduktes kann eine Familien-basierte Verifikation einer Software-Produktlinie durchgeführt werden. Thüm et al. [TAK+14] haben vorgeschlagen zunächst die einzelnen Features zu verifizieren und die Ergebnisse dann bei der Familien-basierten Verifikation zu nutzen. Weitere Arbeiten sollten sich damit auseinandersetzen, wie die Ergebnisse aus der Feature-basierten Verifikation genutzt werden können, um die Verifikation des Metaproduktes zu beschleunigen.

Literaturverzeichnis

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013. (zitiert auf Seite ix, 1, 5 und 6)

[AKGL10] Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Autom. Softw. Eng.*, 17(3):251–300, 2010. (zitiert auf Seite 70)

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Eng.*, 39(1):63–79, 2013. (zitiert auf Seite 2, 8, 9 und 45)

[AL08] Sven Apel and Christian Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In Cesare Pautasso and Éric Tanter, editors, *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2008. (zitiert auf Seite 9)

[ASW+11] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of Feature Interactions using Feature-Aware Verification. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *ASE*, pages 372–375. IEEE, 2011. (zitiert auf Seite 2)

[BCC+03] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Electr. Notes Theor. Comput. Sci.*, 80:75–91, 2003. (zitiert auf Seite 69)

[Ben12] Fabian Benduhn. Contract-Aware Feature Composition. Bachelorarbeit, University of Magdeburg, Germany, 2012. (zitiert auf Seite 17, 18, 31, 33, 45 und 69)

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999. (zitiert auf Seite 15 und 16)

[CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003. (zitiert auf Seite 20)

[CN06] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. The SEI series in software engineering. Addison-Wesley, Boston Mass. u.a, 5. printing edition, 2006. (zitiert auf Seite 5 und 6)

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008. (zitiert auf Seite 2 und 15)

[Ehr02] Wolfgang Ehrenberger. *Software-Verifikation: Verfahren für den Zuverlässigkeitsnachweis von Software*. Hanser, 2002. (zitiert auf Seite 2 und 16)

[KA08] Christian Kästner and Sven Apel. Type-Checking Software Product Lines - A Formal Approach. In *ASE*, pages 258–267. IEEE, 2008. (zitiert auf Seite 70)

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (zitiert auf Seite 6)

[KvRE+12] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward Variability-Aware Testing. In Ina Schaefer and Thomas Thüm, editors, *FOSDGPCE*, pages 1–8. ACM, 2012. (zitiert auf Seite 73)

[LC03] G. Leavens and Y. Cheon. Design by Contract with JML, 2003. (zitiert auf Seite 2, 13, 69 und 74)

[LHB01] Roberto E. Lopez-Herrejon and Don S. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In Jan Bosch, editor, *GCSE*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24. Springer, 2001. (zitiert auf Seite 7)

[Mei13] Jens Meinicke. JML-Based Verification for Feature-Oriented Programming. Bachelorarbeit, University of Magdeburg, Germany, April 2013. (zitiert auf Seite 2, 21, 22, 26, 28, 45, 48, 54, 66 und 70)

[Mey92] Bertrand Meyer. Applying Design by Contract. *Computer (IEEE)*, 25(10):40–51, 1992. (zitiert auf Seite 69)

[OMR10] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In Jan Bosch and Jaejoon Lee, editors, *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2010. (zitiert auf Seite 70)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition, 2005. (zitiert auf Seite 6)

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, pages 419–443, 1997. (zitiert auf Seite 8)

[PS08] Hendrik Post and Carsten Sinz. Configuration Lifting: Verification meets Software Configuration. In *ASE*, pages 347–350. IEEE, 2008. (zitiert auf Seite 2, 20 und 70)

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *ICST*, pages 459–468. IEEE Computer Society, 2010. (zitiert auf Seite 70)

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schäfer, and Saake Gunter. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 2014. To appear; accepted 2014-01-30. (zitiert auf Seite 15, 16, 19, 20, 69 und 74)

[TBA+13] Thomas Thüm, Fabian Benduhn, Sven Apel, André Weigelt, and Gunter Saake. Feature-Oriented Contract Composition. Unveröffentlichtes Manuskript, 2013. (zitiert auf Seite 17, 18, 31 und 33)

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development, 2014. (zitiert auf Seite 2, 7, 45 und 50)

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In Eduardo Santana de Almeida, Tomoji Kishi, Christa Schwanninger, Isabel John, and Klaus Schmid, editors, *SPLC*, pages 191–200. IEEE, 2011. (zitiert auf Seite 6)

[TLD+11] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *Operating Systems Review*, 45(3):10–14, 2011. (zitiert auf Seite 69)

[TSHA12] Thomas Thüm, Ina Schaefer, Martin Hentschel, and Sven Apel. Family-Based Deductive Verification of Software Product Lines. In Klaus Ostermann and Walter Binder, editors, *GPCE*, pages 11–20. ACM, 2012. (zitiert auf Seite 22, 26, 28 und 70)

[TSK+12] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel, and Gunter Saake. Applying Design by Contract to Feature-Oriented Programming. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2012. (zitiert auf Seite 2, 12, 13, 17, 18, 31 und 69)

[TSKA11] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. Proof Composition for Deductive Verification of Software Product Lines. In *ICST Workshops*, pages 270–277. IEEE Computer Society, 2011. (zitiert auf Seite 70)

[vRAR11] Alexander von Rhein, Sven Apel, and Franco Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *Proc. Java Pathfinder Workshop*, page 82, 2011. (zitiert auf Seite 73)

[Wei13] André Weigelt. Methoden-basierte Komposition von Kontrakten in Feature-orientierter Programmierung. Bachelorarbeit, Universität Magdeburg, Germany, August 2013. (zitiert auf Seite 7, 18, 19, 38, 45 und 69)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 24. Februar 2014