Featuremodellbasiertes und kombinatorisches Testen von Software-Produktlinien

Sebastian Oster, Philipp Ritter, Andy Schürr

{oster, pritter, schuerr}@es.tu-darmstadt.de

Abstract: Software-Produktlinien-Entwicklung bietet eine systematische Wiederverwendung von Software-Artefakten. Auf Grund der Tatsache, dass viele Produkte aus einer Produktlinie abgeleitet werden können, ist es unerlässlich Testverfahren zu entwickeln, die zum einen eine möglichst vollständige Abdeckung von allen möglichen Produkten sicherstellen und zum anderen weniger aufwändig sind, als jedes Produkt einzeln zu testen. In diesem Beitrag wird die Entwicklung eines neuen Algorithmus zur Auswahl einer Menge von Produktinstanzen als Testkandidaten beschrieben. Dieser vereint Techniken der Modelltransformation, des kombinatorischen Testens und des Lösens binärer Constraintsysteme vermittels Forward Checking.

1 Einleitung

Software-Produktlinien-Entwicklung ist eine Methode zur systematischen Wiederverwendung von Software-Artefakten. Dabei wird eine Gruppe von Software-Applikationen oder -Produkten beschrieben, die sich gewisse gemeinsame Eigenschaften teilen und somit spezielle Anforderungen einer Domäne erfüllen [PBvdL05].

Software-Produktlinien (SPL) werden in verschiedenen Domänen, unter anderem im Automobil-Bereich, erfolgreich eingesetzt, um die Kosten der Entwicklung und Wartung zu senken und die Qualität der einzelnen Produkte zu steigern. Im Automobil-Bereich werden die Entwickler derzeit mit einer Situation konfrontiert, in der (1) 50-70 ECUs (Electronic Control Units) für die Steuerung diverser Funktionen in einem Auto verbaut werden und (2) jede dieser ECUs bis zu 10.000 verschiedene Konfigurationsmöglichkeiten besitzt. Daraus ergeben sich Millionen von verschiedenen Variationen. Bei gewissen Modellen einiger Hersteller rollen bereits Autos vom Band, die alle unterschiedliche Softwarekonfigurationen enthalten. In der Praxis werden Software-Produktlinien häufig dadurch validiert, dass jede generierte Produktinstanz wie ein einzelnes Softwareprodukt getestet wird [TTK04]. Diese Methode bietet den Vorteil, dass man einen Großteil der Variabilität vernachlässigen und bekannte und bewährte Methoden aus dem Software-Engineering einsetzen kann. Dies ist jedoch in vielen Fällen, wie das Beispiel aus dem Automobil-Bereich zeigt, auf Grund der hohen Anzahl an möglichen Produkten nicht mehr ausführbar. Aus diesem Grund wird nach Lösungen gesucht, die den Testaufwand reduzieren.

Erstrebenswert ist die Entwicklung einer Testmethode, die den Testaufwand minimiert, dabei jedoch nicht auf den Vorteil, den das individuelle Produkttesten mit sich bringt,

verzichtet. Ziel wäre die Identifizierung einer minimalen repräsentativen Menge von Produkten. Da dies ein NP-vollständiges Problem darstellt [Sch07] kann eine solche Menge nur durch Heuristiken approximiert werden. Das erfolgreiche Testen dieser Menge soll mit hoher Wahrscheinlichkeit garantieren, dass der Test aller weiteren Produkte ebenfalls erfolgreich ist.

In dieser Arbeit wird untersucht, inwieweit sich kombinatorisches Testen als Grundlage einer solchen Heuristik eignet und wie kombinatorisches Testen auf Featuremodelle angewendet werden kann. Kombinatorisches Testen wird zur Reduzierung des Testaufwands bei der Auswahl von Testparametern bereits erfolgreich eingesetzt [CDKP94, LT98]. Da prinzipiell Features der Parametrisierung von SPLs dienen, scheint es vielversprechend, kombinatorisches Testen auf SPLs anzuwenden [McG01, CDS07]. Dabei werden Features als Parameter einer SPL interpretiert. Fokus dieses Beitrags ist die Bewältigung der technischen Herausforderungen um kombinatorisches Testen auf Featuremodelle anzuwenden.

Der hier vorgestellte Ansatz wurde im Rahmen des BMBF-Projektes feasiPLe entwickelt [fC06] und ist eine Weiterführung der Arbeit in [OS09]. Das Verfahren wurde auf zwei verschiedene Arten realisiert. Zum einen wurde ein eigener Feature-Modell-Editor implementiert, der die Vorgehensweise auf Basis von Modelltransformation mit Fujaba/MOFLON [AKRS06] realisiert. Zum anderen wurde im Rahmen des feasiPLe-Projekts ein Plugin für das Tool pure::variants der Firma pure-systems erstellt, welches die Methodik umsetzt. In diesem Beitrag wird ausschließlich das pure::variants Plugin beschrieben.

Für die Beschreibung der Vorgehensweise ist die Arbeit wie folgt aufgebaut. In Abschnitt 2 werden themenverwandte Arbeiten aufgeführt und erläutert. Anschließend werden in Abschnitt 3 Grundlagen zu Software-Produktlinien und kombinatorischem Testen beschrieben. Abschnitt 4 widmet sich dem Konzept dieses Beitrags. In diesem werden zum einen die notwendigen Schritte beschrieben, um die Anwendung von kombinatorischem Testen auf Featuremodellen zu ermöglichen und zum anderen der eigens für diese Anwendung entwickelte Algorithmus zur Realisierung des kombinatorischen Testens vorgestellt. Jeder einzelne Schritt wird dabei anhand eines Anwendungsbeispiels veranschaulicht. Eine Beschreibung der Implementierung mit pure::variants ist in Abschnitt 4.3 aufgeführt. Abschließend wird dieser Beitrag in Sektion 5 zusammengefasst und ein Ausblick über weitere Forschungsaktivitäten dargestellt.

2 Verwandte Arbeiten

Testen von Software-Produktlinien

Für das Testen von Software-Produktlinien existieren viele verschiedene Ansätze, mit der Zielsetzung Fehler zu identifizieren, um eine bestimmte Qualität der Software zu gewährleisten. Eine Zusammenfassung ist in [TTK04] aufgeführt. Die Autoren unterteilen die Testansätze in vier Kategorien nach denen Testen in den Entwicklungsprozess eingebunden werden kann.

Product-by-product testing: Als product-by-product testing wird die Testmethodik bezeichnet, die jede einzelne Produktinstanz individuell testet und dabei auf Testverfahren zurückgreift, die in der Software-Engineering-Community bekannt sind. Wie bereits in der Einleitung erwähnt, ist es in vielen Bereichen nicht mehr möglich alle Produkte einzeln zu

testen. Eine Möglichkeit, die Menge an Produkten zu reduzieren, ist die Generierung einer repräsentativen Menge von Produkten für die SPL. Nachteil dieses Ansatzes ist, dass die Bestimmung einer minimalen Testmenge von Produkten ein NP-vollständiges Problem darstellt [Sch07]. In der Regel wird also nicht die minimale Menge von Produkten bestimmt, sondern es werden zu viele, sich äquivalent verhaltende Produktinstanzen erzeugt.
Inkrementelles Testen: In diesem Testverfahren werden, ähnlich wie im product-by-product testing, die Produkte einzeln getestet, jedoch wird der Testaufwand unter Verwendung von Regressionstestverfahren verringert. Dabei wird ein spezielles Produkt basierend auf speziellen Kriterien ausgewählt und ausführlich getestet. Daran anschließend wird ein weiteres Produkt getestet, welches eine sehr hohe Ähnlichkeit im Bezug zu der Funktionalität zum ersten Produkt hat. Die Grundidee ist, dass nur die veränderten Komponenten ausführlich getestet werden müssen, da die Gemeinsamkeiten bereits mit dem ersten Produkt getestet wurden. Eine Schwierigkeit bei dieser Testmethode ist die Tatsache, dass entschieden werden muss, welches Produkt initial ausführlich getestet wird und welche Teile dann nicht mehr getestet werden müssen [Con04, McG01].
Wiederverwendung von Test Assets: Bei dieser Testmethodik steht die Aufteilung in Domain- und Application-Engineering im Vordergrund. Ziel ist es, im Domain-Engineering wiederverwendbare Testartefakte zu entwickeln, die dann für das Testen der Produkte verwendet werden können. Diese Testartefakte werden im Application-Engineering-Prozess an das jeweilige Produkt angepasst. Diesem Ansatz haben sich viele SPL-Testverfahren verschrieben. Bertolino und Gnesi [BG03] erweitern die Category-Patitioning-Methode um Variabilität um diese wiederverwenden zu können. McGregor [McG01] erstellt abstrakte Domänentestfälle, die spezialisiert auf einzelne Applikationen angewendet werden. Ähnlich dazu werden in den Testverfahren CADeT [Oli08] und ScenTED [RKPR05] wiederverwendbare Testmodelle im Domain-Engineering generiert und für jede Variante adaptiert. Nebut et al. generieren Tests auf Basis von Use Cases [NFLTJ04]. Um diese wiederverwenden zu können, wird Variabilität integriert. Weißleder et al. nutzen eine Statemachine um die Funktionalität der gesamten Produktlinie zu modellieren [WSS08]. Diese wird dazu verwendet um produktspezifische Testfälle zu generieren.
Aufteilung nach Verantwortlichkeit: Bei diesem Testverfahren werden den Domain- und Application-Engineering Prozessen die verschiedenen Test-Level (Unit, Integration, System und Acceptance) zugewiesen. Dabei könnten z.B. Unit Tests im Domain-Engineering und die restlichen Test-Level im Application-Engineering ausgeführt werden. Ein weiterer Vorschlag erfolgt in [JhQJ08], in dem für Domain- und Application-Engineering jeweils ein V-Modell mit allen Test-Leveln durchlaufen wird.

Allen Ansätzen ist gemeinsam, dass Algorithmen zur Bestimmung einer Sequenz oder Menge zu testender Produktinstanzen benötigt wird, um nicht alle Produkte testen zu müssen. Eine Arbeit, die sich mit diesem Thema befasst ist [Sch07]. Die Autorin generiert für jede Anforderung eine repräsentative Menge. Ein weiterer Algorithmus wird in dieser Arbeit vorgestellt, welcher auf dem Prinzip des kombinatorischen Testens basiert. Die in diesem Ansatz generierten Produktinstanzen können durch die aufgezählten Methodiken getestet werden.

Kombinatorisches Testen

Um die Korrektheit eines Programms gewährleisten zu können, muss jede mögliche Kombination aller verfügbaren Eingabewerte getestet werden [Bei90]. Dies ist auf Grund der Komplexität der meisten Programme und der daraus resultierenden Kombinationsmöglichkeiten jedoch fast nie realisierbar. Selbst bei einer Software mit lediglich 10 Eingabefeldern mit jeweils 3 Möglichkeiten wären schon $3^{10} = 59.049$ Testfälle abzuarbeiten. Wird die Anzahl der Möglichkeiten auf 5 erhöht, ergeben sich bereits $5^{10} = 9.765.625$ Testfälle. Zahlreiche Konzepte existieren, welche die Testfallanzahl anhand von kombinatorischen Testen minimieren während die Testabdeckung möglichst groß bleibt. Für die Anwendung auf Software-Produktlinien ist es unerlässlich, dass Abhängigkeiten zwischen Parametern, aber auch zwischen Parameterwerten, beachtet werden können. Eine Zusammenfassung einiger solcher Algorithmen ist in [CDS07] gegeben. Dabei wird zwischen mathematischen-, Greedy- und meta-heuristischen Verfahren differenziert. Diese Übersicht zeigt auch, welche dieser Algorithmen in der Lage sind zusätzliche Abhängigkeiten zwischen den Parametern zu verarbeiten. Eine Anwendung des kombinatorischen Testens ist das paarweise Testen. Diese Methode nutzt die Erkenntnis, dass der Großteil aller Softwarefehler aus einzelnen Datenwerten oder der Interaktion zweier Datenwerte resultiert [SM98]. Daher werden lediglich die Eingabefelder paarweise mit jedem anderen Eingabefeld getestet, wobei zwischen diesen zwei Feldern weiterhin alle möglichen Kombinationen gebildet werden. Sehr bekannte Arbeiten sind dabei AETG [CDKP94] oder IPO [LT98]. AETG und IPO erreichen ihr Ziel, alle paarweisen Kombinationen in möglichst wenig Testfällen unterzubringen, auf unterschiedliche Weise, wobei der AETG oftmals eine etwas geringere Testfallanzahl als der IPO erzeugt. Da es sich bei AETG jedoch um ein kommerzielles Projekt handelt, dessen Realisierungsdetails nicht bekannt sind, dient in dieser Arbeit der frei verfügbare IPO als Grundlage für den in Abschnitt 4.2 vorgestellten Algorithmus. Eine kritische Diskussion des paarweisen Testens ist in [BS04] aufgeführt.

3 Grundlagen

Als Anwendungsbeispiel dient ein Ausschnitt einer Handy-Produktlinie, die für die Lehre und Forschung entwickelt wird. Alle Funktionalitäten werden auf Basis der Google Android-Plattform realisiert. Für die Veranschaulichung wird eine Produktlinie gewählt, die über Grundfunktionen, Datenübertragungsmöglichkeiten sowie einige Extras verfügt. Zu den Grundfunktionen zählen das Telefonieren und das Versenden von Nachrichten. Zum Datenaustausch stehen der Produktlinie WLAN, Bluetooth und UMTS zur Verfügung. Mögliche Extras sind ein integrierter MP3-Player und eine Kamera. Diese Produktlinie ermöglicht die Ableitung von 26 gültigen Produktvarianten.

Software-Produktlinien und Featuremodelle

Featuremodelle werden für die explizite Darstellung der Variabilität und der Gemeinsamkeiten innerhalb der SPL-Entwicklung verwendet. Sie bestehen aus hierarchisch angeordneten Features, welche Systemeigenschaften beschreiben, die relevant für einige Stakeholder sind [CHE05]. In [Bos00] werden Features als „eine logische Gruppe von Anforderungen“ definiert. Im Domain-Engineering wird dazu das gesamte Featuremodell

aufgestellt. Im Application-Engineering wird nur noch mit einem Teilbaum des Featuremodells gearbeitet, der ein einzelnes Produkt der SPL beschreibt. Featuremodelle sind intuitiv verständlich und bieten neben der hierarchischen Struktur weitere Notationen und Abhängigkeiten um die Auswahlmöglichkeiten von Features zu kontrollieren. Das erste Featuremodell wurde von Kang et al. [KCH+90] als Teil der FODA-Machbarkeitsstudie eingeführt. In diesem konnten Pflicht-, Optional- und Alternativ-Features definiert werden. Im Laufe der Zeit wurden Erweiterungen für das Featuremodell entwickelt und diskutiert. Einige Erweiterungen werden in [CHE05] erfasst.

In dieser Arbeit wird die Notation aus [KCH+90] verwendet und durch grafische Exclude- und Require-Kanten ergänzt. Exclude-Beziehungen werden als Kante zwischen zwei sich ausschließende Features eingezeichnet. Unidirektionale Require-Kanten verbinden Features, bei denen die Pfeilrichtung angibt, welches Feature das andere für die Erstellung eines Produktes benötigt. Weiterhin wird eine Oder-Gruppe eingeführt. Features innerhalb dieser Oder-Gruppe können in beliebigen Kombinationen in ein Produkt integriert werden. Einzige Restriktion ist, dass mindestens ein Feature dieser Gruppe ausgewählt wird. Abbildung 1 zeigt das Featuremodell der Handy-Produktlinie. Wie bereits erwähnt, bietet

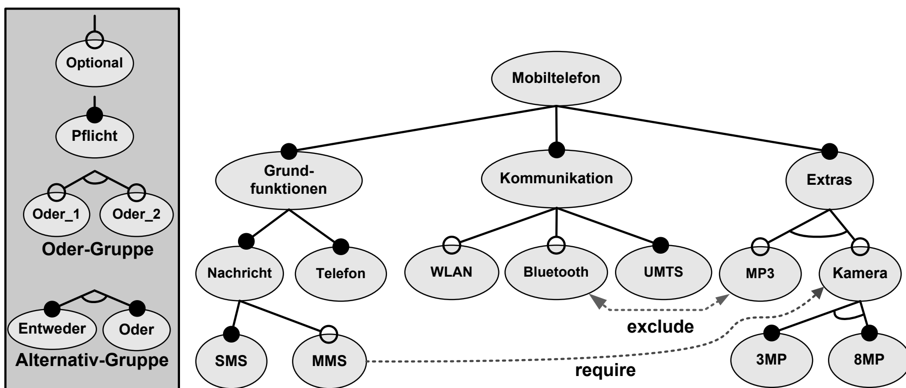

Abbildung 1: Featuremodell des Anwendungsbeispiels

das Handy verschiedene Grundfunktionen, Kommunikations-Funktionalitäten und Extras. Zu den Grundfunktionen zählt das Telefonieren und das Versenden von Nachrichten. Das Telefonieren und das Versenden von SMS ist in jedem Produkt integriert. Das Versenden von MMS ist jedoch optional auswählbar und dementsprechend durch einen unausgefüllten Kreis am Ende der Kante markiert. Für die Kommunikation stehen WLAN und Bluetooth als optionale Features zur Verfügung. UMTS ist in jeder Instanz dieser Produktlinie enthalten. Als Extras stehen ein integrierter MP3-Player und eine integrierte Kamera zur Verfügung. Diese beiden Features sind in einer Oder-Gruppe eingebunden, was zur Folge hat, dass mindestens eines dieser Extras verwendet werden muss. Die Kamera kann entweder einen 3-Megapixel-Sensor oder einen 8-Megapixel-Sensor ansteuern.

Zusätzlich muss bei der Herleitung von Produkten eine Require- und eine Exclude-Abhängigkeit beachtet werden. Wenn MMS-Nachrichten verschickt werden sollen, dann muss eine Kamera integriert werden. MP3-Player und Bluetooth Anbindung schließen sich aus.

Diese Restriktionen sind erdacht, zeigen aber, dass bei Featuremodellen nicht nur „intuitive“ Abhängigkeiten bestehen, sondern durchaus welche, die aus Marketing Sicht Sinn ergeben.

4 Konzept

Features dienen der Parametrisierung von Software-Produktlinien. Die An- und Abwahl von Features entscheidet über die Ausprägung der abgeleiteten Produkte. Daher scheint es vielversprechend zu sein, bekannte Methoden aus dem Software Test auf Software Produktlinien anzuwenden, die sich der Reduzierung des Testaufwands anhand der Parametrisierung widmen [CDS07]. Einer dieser Ansätze ist das kombinatorische Testen, bei dem nur spezielle Kombinationen und nicht alle möglichen Kombinationen von Parametern getestet werden. Es existiert kein Ansatz, der die Anwendung des kombinatorischen bzw. paarweisen Testen auf Featuremodelle beschreibt. Auf Grund der Tatsache, dass Algorithmen des paarweisen Testens Eingabe-Parameter mit dazugehörigen Parameterwerten erwarten, ist es diesen nicht möglich die hierarchische Struktur inklusive Constraints eines Featuremodells zu verarbeiten.

Für die Anwendung des paarweisen Testen auf Featuremodelle sind intuitiv zwei Lösungswege möglich. Die Algorithmen müssen entweder erweitert werden, damit sie die komplexe Struktur und internen Abhängigkeiten eines Featuremodells verarbeiten können oder das Featuremodell muss umstrukturiert werden, sodass die semantische Äquivalenz erhalten bleibt und sich Parameter und Parameterwerte für die Algorithmen ergeben. In dieser Arbeit werden beide Lösungswege kombiniert. Die Struktur des Featuremodells wird durch eine Tiefenreduktion verändert, wobei Parameter und Parameterwerte gebildet werden, auf denen Algorithmen des kombinatorischen Testens arbeiten können. Zudem wird ein Algorithmus entwickelt, ähnlich zum IPO-Algorithmus, der Constraints verarbeiten kann.

4.1 Tiefenreduktion des Featuremodells

Um aus dem Featuremodell Parameter und Parameterwerte zu extrahieren, wurde eine Tiefenreduktion entwickelt, welche aus folgenden zwei Schritten besteht:
1. alle Features mit ihrer zugehörigen Notation werden iterativ nach oben gezogen. Ebenso bleiben die binären Constraints in Form von Require- und Exclude-Kanten erhalten.
2. anschließend werden jedem Feature die korrespondierenden Belegungsmöglichkeiten als Parameterwert angehängt.
Die Tiefenreduktion setzt sich aus mehreren Modelltransformationsregeln zusammen, die das iterative Hochziehen von Features realisieren. Diese Modelltransformationen wurden einerseits mit MOFLON prototypisch realisiert, andererseits als eigenständige Java-Implementierung als Plugin für pure::variants entwickelt. Diese Regeln werden jeweils auf Teilbäume mit ihren assoziierten Constraints angewendet. Diese Teilbäume bestehen immer aus drei Ebenen von Knoten: dem Großvaterknoten, dem Vaterknoten und dem Kind-

knoten. Die Kindknoten werden auf die Ebene des Vaterknotens gehoben, also mit auf die Ebene unter den Großvaterknoten gesetzt. Neben der Hierarchie spielen auch die verschiedenen Notationen der Features eine essentielle Rolle. Optional-Features können nicht genauso hochgezogen werden, wie Pflicht-Knoten usw.. Abbildung 2 zeigt die Hochzieh-

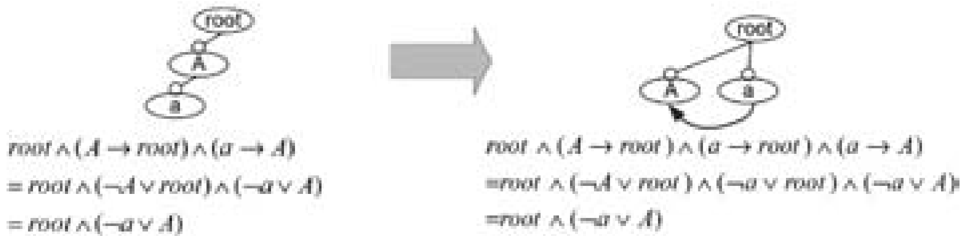

Abbildung 2: Hochziehregel: Optionaler Kindknoten und optionaler Vaterknoten

regel für einen optionalen Kindknoten **a** und einem optionalen Vaterknoten **A**. Die Notation des Großvaterknotens **root** ist zu diesem Zeitpunkt irrelevant. Der Kindknoten kann als optionaler Knoten neben den Vaterknoten gezogen werden. Da jedoch sichergestellt werden muss, dass **a** nur dann ausgewählt werden kann, wenn auch der Vaterknoten **A** ausgewählt ist, wird eine Require-Kante von **a** auf **A** gezogen. Um die semantische Äquivalenz zwischen dem Ursprungs- und dem hochgezogenen Featuremodell zu überprüfen, können beide in einen logischen Ausdruck nach den Regeln in [CW07] übersetzt werden. Dieser Äquivalenznachweis ist ebenfalls in Abbildung 2 aufgeführt.

Für Pflicht-Kindknoten gilt immer die selbe Regel: das Pflicht-Feature wird in den Vaterknoten integriert, da der Vaterknoten nie ohne den Pflicht-Kindknoten existieren kann. Für alle anderen Notationen ergeben sich individuelle Hochziehregeln. Aus jeder Vater-Kind Kombination resultiert eine eigene Regel: 12 Regeln + 1 Regel für Pflicht-Kindkonten → 13 Regeln. Bei der Ausführung der Regeln kommen zusätzliche Require- und Exclude-Kanten hinzu. Diese müssen zusammen mit den ursprünglichen Require- und Exclude-Kanten erhalten bleiben. Diese Regeln werden iterativ ausgeführt, bis der Großvaterknoten dem Wurzelknoten des Featuremodellels entspricht. Angewendet auf das Anwendungsszenario ergibt sich das in Abbildung 3 dargestellte, flache Featuremodell.

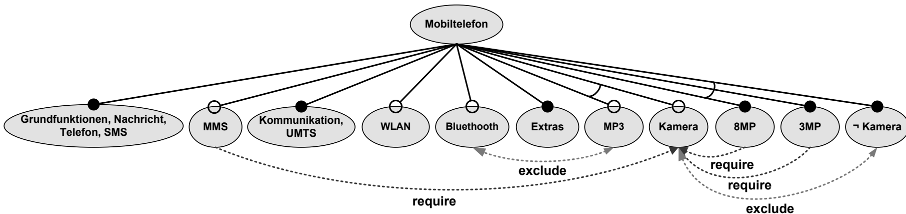

Abbildung 3: Tiefenreduziertes Featuremodell des Anwendungsbeispiels

Ebenso wie bei den einzelnen Regeln kann auch hier zur Sicherheit die semantische Äquivalenz zwischen dem Ursprungs-Featuremodell und dem flachen Featuremodell durch

Übersetzen in einen logischen Ausdruck nachgewiesen werden. Beide Modelle ergeben den selben logischen Ausdruck:

$$\begin{aligned}
&Mobiltelefon \wedge Grundfunktionen \wedge Nachricht \wedge Telefon \wedge SMS \\
&\wedge Kommunikation \wedge UMTS \wedge Extras \wedge (MP3 \vee Kamera) \wedge \\
&\vee [(\neg 3MP \wedge \neg 8MP \wedge \neg Kamera) \vee (3MP \wedge \neg 8MP \wedge Kamera) \\
&\vee (\neg 3MP \wedge 8MP \wedge Kamera)] \wedge (\neg MMS \vee Kamera) \\
&\wedge [(\neg Bluethooth \wedge MP3) \vee (\neg MP3 \wedge Bluetooth)]
\end{aligned} \tag{1}$$

Nach der Tiefenreduktion entspricht das flache Featuremodell einer Menge von Parametern mit ihren zugehörigen Wertebereichen. Jedes Feature wird für das paarweise Testen als ein Parameter behandelt. Zusätzliche Abhängigkeiten müssen dabei beachtet werden. Letzter Schritt für die Vorbereitung ist die Extraktion von Parameterwerten für diese Parameter.

Die Parameterwerte entsprechen den möglichen Belegungen, die ein Parameter einnehmen kann. Ein optionales Feature hat beispielsweise die Werte „vorhanden" und „nicht vorhanden". Pflicht-Knoten haben immer nur einen Wert, nämlich vorhanden. Bei Oder- und Alternativ-Gruppen ist diese Belegung etwas komplexer. In einer Oder-Gruppe sind alle Featurekombinationen möglich, nur gibt es die Einschränkung, dass immer mindestens eines der Features ausgewählt wird. Bei Alternativ-Gruppen muss genau ein Gruppenelement ausgewählt werden wobei die anderen Features ausgeschlossen werden.

Um die Parameterwerte in dem flachen Featuremodell abzubilden, wird eine zusätzliche Ebene eingeführt. In Abbildung 4 ist das flache Featuremodell mit einer zusätzlichen Ebene abgebildet, die die Parameterwerte der Parameter enthält. Die Notation der Parameter wird in Pflicht-Notation umgewandelt, da die Ursprungs-Notation nun durch die Wertebelegungen repräsentiert wird. Das optionale Feature MMS ist nun ein Parameter mit den zwei mögliche Werten MMS und ¬ MMS. Da nur genau einer dieser Wertebelegungen bei der Instanziierung einer Produktvariante und beim Test möglich ist, sind diese Werte als Alternativ-Gruppe markiert.

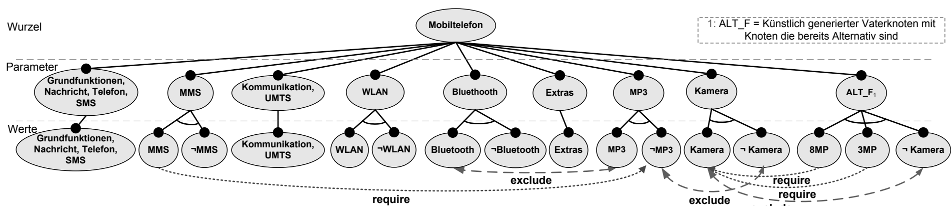

Abbildung 4: Tiefenreduziertes Featuremodell mit Parameterwerten

Für die Erhaltung von Require- und Exclude-Abhängigkeiten müssen diese auf die Parameterwerte übertragen werden. Ein Require-Constraint, welcher zum Beispiel auf das optionale Feature Kamera verweist, muss nun auf den Parameterwert Kamera verschoben werden.

Die Darstellung von Parametern und Parameterwerten skaliert in dieser Darstellungsart nicht sehr gut für große Featuremodelle. Bei der Integration in pure::variants wird dieses Modell nie angezeigt, sondern dient nur für die interne Weiterverarbeitung. Eine Anzeige des flachen Featuremodells ist nicht geplant und auch nicht nötig, da keine weiteren Vorteile aus dieser Darstellung gewonnen werden können. Für die Veranschaulichung von Parametern und Parameterwerten ist die Darstellung durch eine Tabelle vorzuziehen. Eine Überführung des flachen Featuremodells in eine Tabelle ist ohne Weiteres möglich.

4.2 Produktinstanzerzeugung durch paarweises Testen

Nachdem das Featuremodell in Parameter und dazugehörige Werte überführt wurde, können Algorithmen zur Realisierung von paarweisem Testen angewendet werden. Grundidee hinter dem Generierungsalgorithmus ist die Vermutung, dass Fehler in der Integration von Features in 2-er Kombinationen auftreten. Wie der Vergleich in [CDS07] zeigt, können nur wenige Algorithmen zusätzliche Constraints zwischen Parametern und ihren Werten verarbeiten. Diese sind jedoch nicht in ihrer Funktionalität offen gelegt. Daher wurde auf Basis des IPO-Algorithmus [LT98] ein Verfahren entwickelt, welches paarweise Kombinationen unter Beachtung von ggf. existierenden Constraints erzeugt. Der Pseudocode ist in Abbildung 5 aufgeführt. Dazu wird zunächst eine Liste aller möglichen paarweisen

```
01 begin
02   //M is a list of all pairs which have to be covered by a testcase
03   M := {(vi,vk) | vi and vk not exclude each other,
              vi and vk are values of parameters pi and pk | k!=i};
04   T := {};           // list of test cases
05   begin
06     while M != {}    //M is not empty
07     begin
08       P := {(v0,v1) | (v0,v1) ε  M};  //testpair taken from M;
09       t := valid test case that covers at least pair P
              (determined by forward checking);
10       M':= {contains all pairs covered in t};
11       M := M - M';   //remove covered pairs from list
12       T := T + t;    //add new test t to list of tests
13     end
14   end
15 end
```

Abbildung 5: Pseudocode der Produktinstanziierung

Kombinationen erstellt (Zeile 3). Um gültige Produktinstanzen zu bestimmen, wird eine Kombination aus dieser Liste als Startwert einer neuen Produktinstanz verwendet und anschließend mit Hilfe eines Forward-Checking-Algorithmus [HE80] für jeden Parameter ein Wert bestimmt wird (Zeile 9). Eine gültige Produktinstanz liegt vor, wenn für jeden Parameter ein Wert gefunden wurde, der keinen anderen in der Produktinstanz befindlichen Wert durch eine Exclude-Kante ausschließt sowie alle Require-Kanten erfüllt sind. Ist eine solche Produktinstanz generiert worden, werden alle in der Produktinstanz befindlichen

paarweisen Kombinationen (Zeile 10) aus der Liste der benötigten Kombinationen entfernt (Zeile 11) und es werden für die übrig gebliebenen Kombinationen weitere Produktinstanzen erzeugt (Zeile 6). Kann der Forward-Checking-Algorithmus aus dem Startwert keine gültige Produktinstanz erzeugen, wird diese Kombination verworfen.

Das Ergebnis dieses Algorithmus ist eine Menge von Produktinstanzen, die z.B. genutzt werden können, um die SPL durch die Ausführung von Systemtests auf jeder Instanz zu validieren. Die Anwendung des Algorithmus' auf die Handy-SPL resultiert in 8 Produktinstanzen, welche in Abbildung 6 tabellarisch aufgeführt werden.

G,N,T,S	Ko,UMTS	Extras	MMS	WLAN	BT	MP3	Kamera	ALF_F
G,N,T,S	Ko,UMTS	Extras	MMS	WLAN	BT	¬MP3	Kamera	8MP
G,N,T,S	Ko,UMTS	Extras	¬MMS	¬WLAN	¬BT	MP3	¬Kamera	¬Kamera
G,N,T,S	Ko,UMTS	Extras	¬MMS	WLAN	¬BT	¬MP3	Kamera	3MP
G,N,T,S	Ko,UMTS	Extras	MMS	¬WLAN	¬BT	MP3	Kamera	3MP
G,N,T,S	Ko,UMTS	Extras	¬MMS	¬WLAN	BT	¬MP3	Kamera	8MP
G,N,T,S	Ko,UMTS	Extras	¬MMS	WLAN	¬BT	MP3	¬Kamera	¬Kamera
G,N,T,S	Ko,UMTS	Extras	¬MMS	WLAN	BT	¬MP3	Kamera	3MP
G,N,T,S	Ko,UMTS	Extras	¬MMS	WLAN	¬BT	MP3	Kamera	8MP

Abbildung 6: Ergebnis der Produktinstanziierung auf der Handy-SPL

4.3 Integration in pure::variants

pure::variants, ein weit verbreitetes und mächtiges Tool für das Variantenmanagement, speichert Featurebäume in einer XML-Struktur. Somit ist es möglich, durch XML-Parser die von pure::variants erzeugten Featurebäume einzulesen, in eine eigene Datenstruktur zu übertragen und ggf. wieder als ein, für pure::variants lesbares, Featuremodell abzuspeichern. Durch die Unterstützung von pure-systems war es möglich, den in pure::variants integrierten Parser zu nutzen, womit auf die Umsetzung eines eigenen Parsers verzichtet werden konnte. Um die nötigen Transformationen, d.h. die Tiefenreduktion und die Erzeugung der Produktinstanzen, möglichst einfach aufrufen zu können, wurde ein Eclipse-Plugin entwickelt, welches im Kontextmenü der Featurebäume den Aufruf der Transformationen ermöglicht. Die so erzeugten Produktinstanzen werden in einem Unterordner abgelegt und können durch pure::variants betrachtet bzw. weiter verarbeitet werden.

5 Zusammenfassung und Ausblick

Ziel ist, mit Hilfe von kombinatorischem Testen, den Testaufwand für SPLs zu reduzieren. Dabei wird in diesem Beitrag die Grundlage für die Anwendung des kombinatorischen Testens auf ein Featuremodell geschaffen. Es wurde eine Methodik zur Anpassung des Featuremodells vorgestellt, um kombinatorisches Testen anzuwenden. Durch die Tiefenreduktion kann das Featuremodell durch eine einfache Liste innerhalb eines Programms (z.B. pure::variants) dargestellt und verarbeitet werden. Dabei verändert die Tiefenreduktion den Inhalt des Featuremodells nicht sondern ausschließlich die Struktur. Die semantische Äquivalenz kann durch Übersetzen in logische Ausdrücke überprüft werden.

In diesem Beitrag wurde ein Algorithmus für paarweises Testen entwickelt, der sich an

dem IPO-Algorithmus orientiert. Ebenfalls verarbeitet dieser Algorithmus Abhängigkeiten zwischen Parametern und Parameterwerten. Dabei werden zwei Arten von Constraints ausgewertet: bidirektionale Excludes und unidirektionale Requires. Dazu wurde der Algorithmus durch Forward Checking ergänzt, so dass ausschließlich zulässige Produktvarianten bei der paarweisen Kombination gebildet werden. Mit der hier vorgestellten Methode ist es möglich, eine vorliegenden Produktlinie des feasiPLe BMBF-Projekts [fC06] mit 1,8 Mio. Möglichkeiten auf 23 zu testende Produkte zu reduzieren.

Als Basis zur Erstellung einer repräsentativen Testmenge scheint das paarweise Testen vielversprechend zu sein. Exemplarische Studien unter Verwendung von pure::variants haben gezeigt, dass eine 100%ige Abdeckung aller paarweisen Kombinationen erreicht wird und Fehler in 3er-Kombinationen zu ca. 70-80% gefunden werden.

Weiterhin werden in zukünftigen Arbeiten Systemabhängigkeiten mit berücksichtigt, um unabhängige Teilbäume zu identifizieren, so dass nicht mehr alle Feature miteinander paarweise kombiniert werden, sondern nur noch voneinander abhängige. Ein erster Ansatz wurde in [OS09] erarbeitet und eine Evaluation ist Fokus aktueller Forschungsarbeiten. Zusätzlich werden in noch folgenden Beiträgen den Features Attribute zugewiesen, die den Kosten, Gewinn und Testaufwand beschreiben um die Auswahl der Varianten zusätzlich zu beinflussen. Features, die oft genutzt oder sicherheitskritisch sind, können so mit Priorität getestet werden. Vorteil dieses Ansatzes ist, dass Benutzungsszenarien und Anforderungen mit berücksichtigt werden können, die bestimmten Varianten einen höheren Stellenwert zuordnen als anderen Varianten [BC05].

Sollte sich in weiteren Arbeiten herausstellen, dass die Anwendung des kombinatorischen Testens auf SPLs nicht den erwarteten Erfolg bringt, so wurde in dieser Arbeit jedoch ein Algorithmus entwickelt der das paarweise Testen mit Berücksichtigung von Constraints und Forward Checking realisiert.

Literatur

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke und A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture - Foundations and Applications: Second European Conference*, Jgg. 4066 of *Lecture Notes in Computer Science (LNCS)*, Seiten 361–375, 2006.

[BC05] Renée C. Bryce und Charles J. Colbourn. Test prioritization for pairwise interaction coverage. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[Bei90] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[BG03] A. Bertolino und S. Gnesi. Use Case-based Testing of Product Lines. *SIGSOFT Software Engineering Notes*, 28(5):355–358, 2003.

[Bos00] Jan Bosch. *Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley Longman, Amsterdam, 2000.

[BS04] J. Bach und P. Shroeder. Pairwise testing: A best practice that isn't. In *Proceedings of 22nd Pacific Northwest Software Quality Conference*, Seiten 180–196. Citeseer, 2004.

[CDKP94] D. M. Cohen, S. R. Dalal, A. Kajla und G.C. Patton. The Automatic Efficient Tests Generator. *Fifth Int'l Symposium on Software Reliability Engineering*, IEEE:303–309, 1994.

[CDS07] M.B. Cohen, M.B. Dwyer und J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Intl. symp. on Software testing and analysis*, Seiten 129–139, 2007.

[CHE05] K. Czarnecki, S. Helsen und U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. Seiten 143–169, 2005.

[Con04] C. Condron. A Domain Approach to Test Automation of Product Lines. In *SPLiT 2004 Proceedings*, Seiten 27–35, 2004.

[CW07] K. Czarnecki und A. Wasowski. Feature diagrams and logics: There and back again. In *Software Product Line Conference*, Seiten 23–34, 2007.

[fC06] feasiPLE Consortium. www.feasiple.de. 2006.

[HE80] R.M. Haralick und G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[JhQJ08] L. Jin-hua, L. Qiong und L. Jing. The W-Model for Testing Software Product Lines. In *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, Jgg. 1, Seiten 690–693, 2008.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak und A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Bericht, Carnegie-Mellon U. SEI, 1990.

[LT98] Y. Lei und K.C. Tai. In-parameter-order: a test generation strategy for pairwise testing. In *IEEE High Assurance Systems Engineering Symposium*, Seiten 254–261, 1998.

[McG01] J. D. McGregor. Testing a Software Product Line. Bericht CMU/SEI-2001-TR-022, 2001.

[NFLTJ04] C. Nebut, F. Fleurey, Y. Le Traon und J.M. Jézéquel. A Requirement-Based Approach to Test Product Families. *LNCS, Springer-Verlag*, Seiten 198–210, 2004.

[Oli08] E. M. Olimpiew. *Model-Based Testing for Software Product Lines*. Dissertation, George Mason University, 2008.

[OS09] S. Oster und A. Schürr. Architekturgetriebenes Pairwise-Testing für Software-Produktlinien. In *Workshop Software Engineering 2009: PVLZ 2009*, March 2009.

[PBvdL05] K. Pohl, G. Boeckle und F. van der Linden. *Software Product Line Engineering*. Springer-Verlag, 2005.

[RKPR05] A. Reuys, E. Kamsties, K. Pohl und S. Reis. Model-based System Testing of Software Product Families. In *CAiSE*, Seiten 519–534, 2005.

[Sch07] Kathrin Scheidemann. Verifying Families of System Configurations. *Doctoral Thesis*, TU Munich, 2007.

[SM98] B. Stevens und E. Mendelsohn. Efficient software testing protocols. In *Conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1998.

[TTK04] A. Tevanlinna, J. Taina und R. Kauppinen. Product Family Testing: a Survey. *ACM SIGSOFT Software Engineering Notes.*, 29, 2004.

[WSS08] S. Weißleder, D. Sokenou und B. Schlinglo. Reusing State Machines for Automatic Test Generation in Product Lines. In *MoTiP Workshop*, 2008.