View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Qucosa - Publikationsserver der Universität Leipzig

Universität Leipzig

Wirtschaftswissenschaftliche Fakultät

Institut für Wirtschaftsinformatik

Professur Softwareentwicklung für Wirtschaft und Verwaltung

Prof. Dr. Ulrich W. Eisenecker

Max Lillack, M.Sc

Thema

Variabilitätsextraktion aus makrobasierten Software-Generatoren

Masterarbeit zur Erlangung des akademischen Grades
Master of Science – Wirtschaftsinformatik

vorgelegt von: David Baum

Prüfungsnummer: 90362

Matrikelnummer: 1602334

Email-Adresse: david.baum@naraesk.eu

Telefonnummer: +49 174 58 70 69 1

Anschrift: Lutherstraße 12

08451 Crimmitschau

Leipzig, den 07. Januar 2014

Abstract

Die vorliegende Arbeit beschäftigt sich mit der Frage, wie Variabilitätsinformationen aus den Quelltext von Generatoren extrahiert werden können. Zu diesem Zweck wurde eine Klassifizierung von Variablen entwickelt, die im Vergleich zu bestehenden Ansätzen eine genauere Identifikation von Merkmalen ermöglicht. Zudem bildet die Unterteilung die Basis der Erkennung von Merkmalinteraktionen und Cross-tree-Constraints. Weiterhin wird gezeigt, wie die gewonnenen Informationen durch Merkmalmodelle dargestellt werden können. Da diese auf dem Generator-Quelltext basieren, liefern sie Erkenntnisse über den Lösungsraum der Domäne. Es wird sichtbar, aus welchen Implementierungskomponenten ein Merkmal besteht und welche Beziehungen es zwischen Merkmalen gibt. Allerdings liefert ein automatisch generiertes Merkmalmodell nur wenig Erkenntnisse über den Lösungsraum. Außerdem wurde ein Prototyp entwickelt, der eine Automatisierung des beschriebenen Extraktionsprozesses ermöglicht.

Schlüsselwörter

Merkmalmodelle, Variabilität, Variabilitätsextraktion, Software-Produktlinien

Inhaltsverzeichnis

Inhaltsverzeichnis I

Abbildungsverzeichnis III

Tabellenverzeichnis IV

Listings V

Abkürzungsverzeichnis VI

1 Einleitung 1
- 1.1 Motivation . . . 1
- 1.2 Zielstellung . . . 3
- 1.3 Aufbau der Arbeit . . . 3

2 Software-Produktlinien 5
- 2.1 Definition und Einführung . . . 5
- 2.2 Generative Softwareentwicklung . . . 6
- 2.3 Entwicklungsprozess . . . 7
  - 2.3.1 Vorgehensweisen . . . 7
  - 2.3.2 Domain Engineering . . . 8
  - 2.3.3 Application Engineering . . . 9
- 2.4 Merkmalorientierte Programmierung . . . 10
- 2.5 Beziehungen zwischen Merkmalen . . . 11

3 Variabilität 13
- 3.1 Definitionen . . . 13
- 3.2 Dimensionen . . . 14
- 3.3 Variabilitätsmodellierung . . . 15
  - 3.3.1 Definition . . . 15
  - 3.3.2 Ansätze zur Variabilitätsmodellierung . . . 16
  - 3.3.3 Merkmalmodelle . . . 17
  - 3.3.4 Constraints-Modellierung . . . 20

4 Application Development System 22
- 4.1 Aufbau und Ablauf . . . 22
- 4.2 Sprachkonstrukte . . . 23
  - 4.2.1 Kontrollflussstrukturen . . . 23
  - 4.2.2 Wiederholungsanweisungen . . . 24
  - 4.2.3 Variablen und Parameter . . . 25
- 4.3 Implementierung von Konfigurationswissen . . . 26

5 Variabilitätsextraktion 29
- 5.1 Reverse Engineering . . . 29
- 5.2 Variabilität in Generatorsystemen . . . 29
- 5.3 Bestehende Ansätze . . . 31
- 5.4 Übertragung und Erweiterung bestehender Ansätze auf Generatorsysteme . . . 33

5.5 Statische Code-Analyse 34
5.5.1 Abstrakter Syntaxbaum 34
5.5.2 Kontrollflussanalyse 36
5.5.3 Datenflussanalyse 38
5.5.4 Bedeutung von Variablen 40
5.6 Merkmalerkennung 43
5.7 Erkennen von Merkmalkardinalitäten 46
5.8 Erkennen von Merkmalinteraktionen 48
5.9 Benennung von Merkmalen 50

6 Anwendung **53**
6.1 Empirische Überprüfung 53
6.2 Implementierung 54
6.2.1 Funktionsumfang 54
6.2.2 FeatureIDE 54
6.2.3 Funktionsweise 55
6.3 Evaluation 55

7 Schlussbetrachtung **59**

Literaturverzeichnis **VII**

Abbildungsverzeichnis

2.1 Generatives Domänenmodell 6
2.2 Zusammenhang zwischen investierten Aufwand in Domain und Application Engineering 10

3.1 Variationspunkt 14
3.2 Statische und dynamische Variabilität 15
3.3 Variabilitätstrichter 16
3.4 Merkmalmodell 19
3.5 Kardinalitätsbasiertes Merkmalmodell 19
3.6 Wahrscheinlichkeitsbasiertes Merkmalmodell 20

5.1 Variabilität in Generatorsystemen 30
5.2 Kontrollflussgraph 37
5.3 Kontrollflussgraph einer Entscheidungstabelle 38
5.4 Datenflussgraph 39
5.5 Sequenzen eines Kontrollflussgraphen 45
5.6 Transformation alternativer Sequenzen 46
5.7 Transformation optionaler Sequenzen 47
5.8 Transformation zweier optionaler Sequenzen 47
5.9 Transformation einer Mehrfachverzweigung 48
5.10 Transformation eines delokalisierten Merkmals 49
5.11 Delokalisiertes Merkmal mit unterschiedlichen Variabilitätsinformationen . 51

6.1 FeatureIDE-Merkmalmodell 57

Tabellenverzeichnis

4.1 Entscheidungstabelle 24

6.1 Zuordnungen von Code-Abschnitten zu Merkmalen und Merkmalinteraktionen 57

Listings

4.1 Implementierung einer Entscheidungstabelle 24
4.2 Schleife 25
4.3 Parameter 25
4.4 Variablen 26
4.5 Merkmalinteraktionen mittels geschachtelter `IF`-Abfragen 27
4.6 Merkmalinteraktionen mittels Entscheidungstabelle 27
4.7 OR-Interaktion mittels Entscheidungstabelle 28
4.8 Constranits mittels Entscheidungstabelle 28

5.1 *Version selection by #ifdef* mittels Präprozessoranweisungen 32
5.2 Ausschnitt eines AST einer IF-Anweisung 35
5.3 Merkmalvariablen 41
5.4 Implementierung einer Merkmalinteraktion 42
5.5 Hilfsvariablen 42
5.6 Implementierung einer OR-Constraint 50

6.1 Makro mit mehreren Merkmalen und Merkmalinteraktionen 56

Abkürzungsverzeichnis

ADS	Application Development System
AmAVaG	Automatische Architektur- und Variabilitätsanalyse in Generatorsystemen
AST	Abstract Syntax Tree
COBOL	Common Business Oriented Language
CVL	Common Variability Language
DAG	Directed Acyclic Graph
DFS	Depth-first Search
DSL	Domain Specific Language
FOP	Feature-oriented Programming
OR	Inklusives Oder
PDL	Processor Development Language
PL/I	Programming Language One
SAT-Solver	Satisfiability-Solver
SPL	Software-Produktlinie
TXL	Turing Extender Language
UML	Unified Modeling Language
XML	Extensible Markup Language
XOR	Exklusives Oder

1 Einleitung

1.1 Motivation

Die Arbeit an Softwaresystemen endet nicht mit der ersten Veröffentlichung, also dem Übergang in den Produktivbetrieb. Stattdessen finden parallel sowohl Weiterentwicklungen als Wartungsprozesse statt. Erstere fügen neue Funktionalitäten hinzu, Letztere beheben beispielsweise aufgetretene Fehler oder nehmen Anpassungen an geänderte technische Rahmenbedingungen vor. Insbesondere betriebliche Softwaresysteme verfügen häufig über einen sehr langen Lebenszyklus, der mehrere Jahrzehnte umfassen kann. Dies führt dazu, dass möglicherweise keiner der ursprünglichen Entwickler mehr verfügbar ist, wenn Änderungen am Quelltext vorzunehmen sind. Neuen Entwicklern steht allerdings in vielen Fällen nur eine unzureichende Dokumentation zur Verfügung. Diese kann beispielsweise nur einen Teil des Systems abdecken oder einen älteren Stand beschreiben. Einst vorhandenes Wissen über die Funktionsweise der Software, über getroffene Entwurfsentscheidungen und dergleichen mehr ist somit verloren gegangen und muss neu gewonnen werden.

Es ist natürlich möglich, all die notwendigen Entwicklungsartefakte wie Klassenmodelle erneut manuell zu erzeugen. Dieser Weg ist jedoch langwierig und aufwendig. Zudem setzt diese Methode Kenntnisse über die Implementierung voraus, die aber eben nicht mehr vorhanden sind. Aus diesem Grund werden automatische Verfahren angewandt, die dabei helfen sollen, das Verständnis über Altsysteme zu erhöhen. Im Rahmen des Forschungsprojekts *Automatische Architektur- und Variabilitätsanalyse in Generatorsystemen (AmAVaG)* werden auf der Makrosprache Application Development System (ADS) basierende Altsysteme untersucht. Dabei handelt es sich um Systeme, die Software generieren. Diese erzeugen dann in Abhängigkeit von einer zuvor durchgeführten Konfiguration Anwendungssysteme. Sollen diese verändert werden, beispielsweise in Form einer Funkionserweiterung, muss der Quelltext des Generators entsprechend verändert und eine Regenerierung durchgeführt werden. Es ist zwar möglich, aber nicht sinnvoll, jedes erzeugte System einzeln zu untersuchen und für jedes beispielsweise ein eigenes Klassendiagramm anzufertigen. Stattdessen soll der Quelltext des Generators im Fokus stehen, da er Informationen über jedes erzeugbare System enthält. Die vorliegende Arbeit soll daher zeigen, wie aus dem Generator die Informationen gewonnen werden können, die zum Verständnis und der Wartung des vorhandenen Quelltextes erforderlich sind. Durch verschiedene Methoden des *Reverse Engineerings* wird versucht, das Verständnis für den vorhandenen Quelltext zu erhöhen.

Seit einigen Jahren hat in der Software-Entwicklung der Ansatz sogenannter Software-Produktlinien (SPL) zunehmende Verbreitung gefunden. Dabei werden einander ähnliche Produkte, die sowohl über Gemeinsamkeiten als auch Unterschiede verfügen, nicht mehr als unabhängige Einzelsysteme entwickelt, sondern sie werden als Mitglieder einer Produktlinie angesehen. Die zur Verfügung stehenden ADS-Programme wurden nicht als SPLs entwickelt. Nichtsdestoweniger handelt sich aber um *Generatorsysteme*. Darunter sollen Softwaresyste-

me verstanden werden, die generierende Techniken einsetzen. Dies trifft auf ADS-Makros zu, da diese der Generierung von Anwendungsprogrammen dienen. Im Gegensatz zu Generatorsystemen liegt SPLs ein sogenanntes *generatives Domänenmodell* zugrunde, das in Abschnitt 2.2 vorgestellt wird. Dazu zählt unter anderem eine Trennung in Problem- und Lösungsraum sowie Konfigurationswissen. Im Rahmen des Forschungsprojekts sollen durch moderne Perspektiven und Methoden neue Erkenntnisse über die Altsysteme gewonnen werden. Einerseits soll dies zur Redokumentation beitragen. Andererseits soll die Interpretation der Makros auf Basis des generativen Domänenmodells erfolgen, Durch diese Vorarbeit soll eine Migration beziehungsweise Modernisierung der ADS-Makros erleichtert werden (vgl. [DSTG 2013]).

Im Laufe des Software-Entwicklungsprozesses werden eine Reihe unterschiedlicher Diagramme, Modelle und weiterer Artefakte erstellt, die alle verschiedene Aspekte darstellen und daher eigene Anwendungsfälle besitzen. Zu diesen gehören auch die Merkmalmodelle (engl. *feature models*). Sie drücken Gemeinsamkeiten und Unterschiede, auch Variabilität genannt, einer SPL aus. Beide werden in Form sogenannter Merkmale dargestellt. Darunter können sich elementare Eigenschaften vorgestellt werden, die unter anderem zur Beschreibung kundenspezifischer Anwendungssysteme dienen. Das Merkmalmodell enthält zum einen die Informationen darüber, welche Bestandteile zu einem Merkmal zusammengefasst wurden. Zum anderen umfasst das Modell Beziehungen zwischen Merkmalen, beispielsweise können sich diese gegenseitige bedingen oder ausschließen.

Im Rahmen dieser Arbeit sollen daher Merkmalmodelle aus dem Quelltext des Generators extrahiert werden. Dieser Vorgang wird als Variabilitätsextraktion bezeichnet. Merkmalmodelle liefern einen Beitrag zum Verständnis der Funktionsweise des Generators. Dies kann helfen, Änderungen an ihm vorzunehmen und deren Wirkungen zu kennen. Darüber hinaus können Merkmalmodelle auch dazu beitragen, die ursprüngliche Anwendungsdomäne besser zu verstehen.

Es gibt bereits eine Reihe von Arbeiten, in denen verschiedene Ansätze zur automatischen Variabilitätsextraktion entwickelt wurden. Die meisten davon konzentrieren sich jedoch auf andere Artefakte als die vorliegende Arbeit. Beispielsweise wird häufig auf vorliegende Spezifikationen zurückgegriffen. Dabei handelt es sich um Dateien, die enthalten, welche Merkmale ein Benutzer ausgewählt hat. Aus ihnen werden die Merkmale und ihre eventuell vorhanden Abhängigkeiten extrahiert. Es existieren jedoch auch Ansätze, die eine Auswertung des Generatorquelltexts vornehmen. Insbesondere sei hier auf die Arbeit von Snelting verwiesen, in der die in C++-Präprozessoranweisungen enthaltenen Variabilitätsinformationen untersucht werden. Die vorliegende Arbeit setzt hier an und überträgt diese Methode auf ADS-Makros, die komplexer sind als die untersuchten Präprozessoranweisungen. Denn beispielsweise werden Letztere in den untersuchten Fällen ausschließlich zur Implementierung von Variabilität verwendet. ADS-Makros hingegen nehmen noch andere Aufgaben wahr, sodass nicht jede Anweisung für die Erzeugung eines Merkmalmodells berücksichtigt werden

muss. Durch diese Arbeit soll daher auch ein Beitrag zu der Frage geleistet werden, wie die für die Variabilitätsextraktion relevanten Elemente identifiziert werden können.

1.2 Zielstellung

Ein Generatorsystem besteht aus drei Hauptartefakten: Dem Generator selbst, den Konfigurationen sowie den erzeugten Produkten. Für diese Arbeit stehen jedoch ausschließlich Ausprägungen der zuerst genannten Artefaktklasse zur Verfügung und zwar in zwei Varianten: Zum einen liegt der Quelltext der Makros vor, zum anderen steht bereits ein Werkzeug zur Verfügung, das ADS-Code in einen abstrakten Syntaxbaum (engl. *Abstract Syntax Tree, AST*) überführen kann. Daher wird sich auf die Frage beschränkt, wie Variabilitätsinformationen aus dem Generator selbst extrahiert werden können. Hierzu sollen zunächst verschiedene Ansätze betrachtet werden, die bisher in der wissenschaftlichen Literatur vorgeschlagen wurden, unter anderem Snelting [1996] und Loesch/Ploedereder [2007]. Zudem soll ihre Eignung für den vorliegenden Anwendungsfall überprüft werden. Sofern sich diese Methoden nicht als geeignet erweisen, sollen sie auf geeignete Weise erweitert werden. Insbesondere gehen bisherige Ansätze davon aus, dass der untersuchte Quelltext ausschließlich der Implementierung von Variabilität dient. Dies ist im Falle der ADS-Makros nicht der Fall. Es soll daher ein Weg gefunden werden, die Anweisungen eindeutig zu identifizieren, die für die Variabilitätsmodellierung relevant sind.

Eine empirische Überprüfung der vorgeschlagenen Regeln kann im Rahmen dieser Arbeit jedoch nicht erfolgen. Es stehen weder Domänenexperten noch geeignete ADS-Makros zur Verfügung, um die Eignung anhand praktischer Beispiele überprüfen zu können. Es soll jedoch eine prototypische Implementierung des beschriebenen Ansatzes zur Variabilitätsextraktion erfolgen. Um den Prototyp innerhalb eines angemessenen Zeitrahmens umsetzen zu können, wurden einige Einschränkungen hinsichtlich des unterstützten Sprachumfangs getroffen. Hierauf wird in Abschnitt 6.2.1 näher eingegangen.

Mithilfe des Prototyps soll eine beispielhafte Evaluation erfolgen. Zu diesem Zweck wird aus einem ADS-Makros das entsprechende Merkmalmodell erzeugt. Anschließend wird exemplarisch gezeigt, wie die gewonnenen Informationen interpretiert werden können und welche Rückschlüsse sie auf das Ausgangsmakro zulassen.

1.3 Aufbau der Arbeit

Im Rahmen dieser Arbeit sollen die theoretischen Grundlagen der Variabilitätsextraktion erläutert werden. Dies soll stets im Kontext von SPLs geschehen, da Variabilität zu deren Haupteigenschaften zählt. Im folgenden Kapitel wird daher auf SPLs eingegangen. Im Vordergrund stehen dabei zum einen die verschiedenen Schritte des Entwicklungsprozesses und zum anderen generative Systeme. Bei Letzteren handelt es sich um eine technische Umsetzungsmöglichkeit einer SPL.

In Kapitel 3 wird mit der Variabilität ein Charakteristikum von Generatorsystemen und SPLs noch einmal separat betrachtet. Dazu gehört eine Differenzierung des Begriffs entlang verschiedener Dimensionen, die im Abschnitt 3.2 behandelt werden. Ein wichtiges Teilgebiet der Variabilität stellt die Variabilitätsmodellierung dar, mit der sich Abschnitt 3.3 auseinandersetzt. Es wird ein kurzer Überblick über die verschiedenen Ansätze gegeben. Detailliert wird auf Merkmalmodelle eingegangen, da sie später genutzt werden, um die extrahierten Variabilitätsinformationen darzustellen. Intensiv diskutiert werden insbesondere die sogenannten *Constraints*, da sie in verwandten Arbeiten oft nachrangig behandelt werden.

Im darauf folgenden Kapitel 4 wird ein konkretes Generatorsystem detailliert behandelt. Es werden die verschiedenen Kontrollflussstrukturen ebenso detailliert betrachtet wie Variablen und Parameter. Zudem wird gezeigt, wie Konfigurationswissen mit den Möglichkeiten von ADS implementiert werden kann. Dies ist notwendig, da der spätere Extraktionsprozess zu Teilen sprachabhängig ist.

Nachdem die Variabilität thematisiert wurde, erfolgt in Kapitel 5 die Hinwendung zur Variabilitätsextraktion. Zunächst erfolgt eine Einordnung in das übergeordnete Konzept des *Reverse Engineerings*. Im Anschluss daran wird gezeigt, welche Möglichkeiten der Variabilitätsextraktion im Kontext generativer Systeme überhaupt existieren, bevor einige bestehende Ansätze vorgestellt werden. Im darauf folgenden Abschnitt wird versucht, diese auf das Szenario der vorliegenden Arbeit zu übertragen. Da sich jedoch kein Ansatz als hinreichend herausstellt, werden bestehende Verfahren erweitert, womit sich Abschnitt 5.4 auseinandersetzt. Nach der Beschreibung der Funktionsweise des gewählten Extraktionsprozesses, wird anhand einiger Kontroll- und Datenflussstrukturen gezeigt, wie sie sich in ein Merkmalmodell überführen lassen.

Anschließend wird in Abschnitt 6.2 der Prototyp beschriebenen, der im Rahmen dieser Arbeit ebenfalls entwickelt wurde. Er dient zur praktischen Durchführung der zuvor beschriebenen Variabilitätsextraktion. Es wurden jedoch einige Einschränkungen getroffen, die zusammen mit den Anforderungen in Abschnitt 6.2.1 festgehalten sind. Der Hauptteil der Arbeit wird durch eine Evaluation eines Beispielmakros abgeschlossen. Dabei wird gezeigt, welche konkreten Informationen aus einem generierten Merkmalmodell abgeleitet werden können.

In der Schlussbetrachtung erfolgt eine Zusammenfassung über die in den vorhergehenden Kapiteln gewonnen Erkenntnisse. Weiterhin werden die Grenzen des bisherigen Ansatzes noch einmal verdeutlicht. Daraus werden einige Problemfelder abgeleitet, deren Bearbeitung hilfreich wäre, um den Prozess der Variabilitätsextraktion weiter zu verbessern.

2 Software-Produktlinien

2.1 Definition und Einführung

In den 1990er Jahren wurde in der Fertigungsindustrie unter dem Begriff *Mass Customization* angestrebt, individualisierte Produkte mit den Verfahren der Massenproduktion herzustellen und dadurch zum gleichen Preis wie herkömmliche Produkte anbieten zu können (vgl. [Piller 1997, 3f]). Beispielsweise stellen moderne Smartphones eine solche Produktlinie dar, da es sie oft in verschiedenen Ausführungen gibt. Gewählt werden kann zum Beispiel zwischen verschiedenen Farben, ob das Modell über eine *GPS*- oder *LTE*-Anbindung verfügen soll und welches Display verbaut wird. SPLs stellen eine Übertragung dieses Ansatzes auf die Software-Entwicklung dar (vgl. [Pohl et al. 2005, 4]):

„*A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* “ [Clements/Northrop 2007, 5]

Das Ziel besteht also in der effizienten Entwicklung einander ähnlicher Software-Produkte. Diese werden jedoch nicht getrennt voneinander entwickelt, sondern in einem gemeinsamen Entwicklungsprozess mit einem hohen Anteil systematischer Wiederverwendung (vgl. [Pohl et al. 2005, 20]). Auch in der herkömmlichen Entwicklung von Einzelsystemen werden Bestandteile „im Kleinen“ wiederverwendet, dazu zählen beispielsweise Algorithmen, Funktionen und Klassen (vgl. [Clements/Northrop 2007, 91]). Bei einer SPL stellt stattdessen jedes Produkt eine Kombination aus zuvor definierten und implementierten Komponenten dar, es besteht somit zu großen Teilen aus wiederverwendeten Entitäten (vgl. [Acher 2011, 10]). In Anlehnung an das obige Beispiel kann zum Beispiel auch das Betriebssystem des Smartphones, in Form einer SPL entwickelt werden. Auch hier sind verschiedene Ausführungen denkbar, die sich zum Beispiel in der Hardware-Unterstützung sowie Software-Ausstattung unterscheiden können. Allerdings sind SPLs oft sehr viel umfangreicher als in diesem Beispiel. So kann eine Automotive-Software über mehrere tausend Ausführungen verfügen (vgl. [Pohl et al. 2005, 87]). Anhand des gegebenen Beispiels ist dennoch bereits eine wesentliche Charakteristik erkennbar: Werden verschiedene Produkte einer SPL verglichen, so sind sowohl Gemeinsamkeiten als auch Unterschiede zu finden. Letztere werden auch als Variabilität bezeichnet. Die systematische Behandlung gemeinsamer und variabler Merkmale stellen ein zentrales Konzept bei der Entwicklung von SPLs dar.

Mit der eben gegebenen Definition sind eine Reihe unterschiedlicher Methoden und Techniken zur Realisierung von SPLs vorstellbar. Die Zusammensetzung der Komponenten kann beispielsweise manuell erfolgen oder vollständig automatisiert ablaufen. Weiterhin können verschiedene Programmierparadigmen eingesetzt werden, um SPLs zu implementieren. Im Folgenden soll daher eine möglichst allgemeingültige Betrachtung des Entwicklungsprozes-

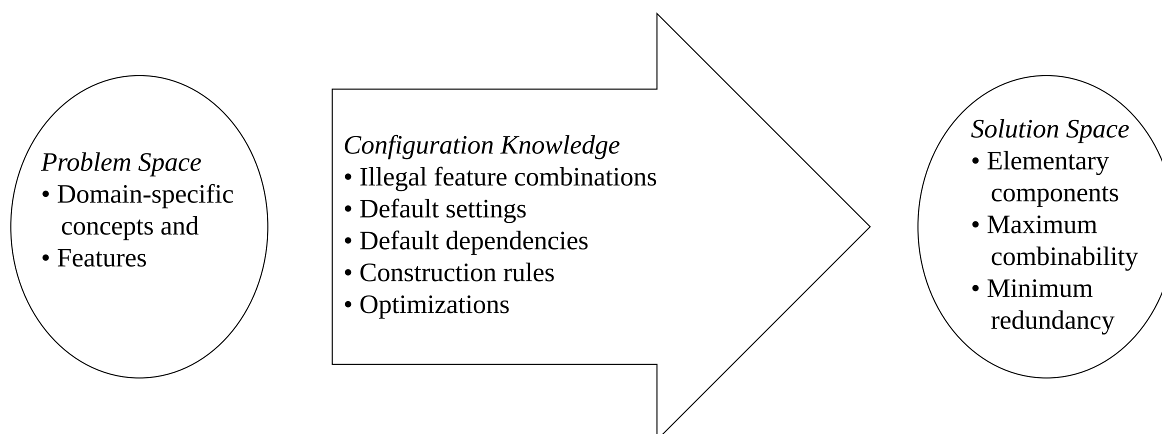

Abbildung 2.1: Generatives Domänenmodell (in Anlehnung an [Czarnecki/Eisenecker 2000, 132])

ses erfolgen. Zudem wird mit der generativen Softwareentwicklung eine Methode zur Entwicklung von SPLs vorgestellt.

2.2 Generative Softwareentwicklung

Die generative Softwareentwicklung stellt einen weit verbreiteten Ansatz dar, um SPLs zu entwickeln und ermöglicht eine weitgehende Automatisierung der Produkterzeugung, dem sogenannten *Application Engineerings*. Die Grundlage bildet das sogenannte generative Domänenmodell, das in Abbildung 2.1 dargestellt ist.

In der generativen Softwareentwicklung wird davon ausgegangen, dass der Benutzer ein Fachanwender ist und nicht über umfangreiches Wissen über die Realisierung der SPL verfügt. Aus diesem Grund erfolgt eine Unterteilung in den Problem- und den Lösungsraum. Ersterer nimmt somit eine fachliche Sicht ein, da das Problem immer in der jeweiligen Domäne angesiedelt ist. Entsprechend werden auch die jeweiligen Fachbegriffe des Anwendungsgebiets verwendet. Der Fachanwender kann nun in den ihn bekannten Termini die Anforderungen an das gewünschte Produkt spezifizieren. Dieser Vorgang wird auch als Spezifikation (des Produkts) bezeichnet. Erneut gibt es zahlreiche technische Möglichkeiten diesen Prozess auszugestalten, etabliert hat sich vor allem die Verwendung einer Domain Specific Language (DSL). Dabei handelt es sich um eine problemraumspezifische Sprache, die exakt für den jeweiligen Anwendungsfall zugeschnitten wurde. Dadurch ist es auf einfache Weise möglich, zu spezifizieren, über welche Eigenschaften das gewünschte Produkt verfügen soll (vgl. [Czarnecki/Eisenecker 2000, 138f]).

Der Generator ist das zentrale Element eines generativen Systems. Auf Basis der Spezifikation erstellt er die Konfiguration. Dabei handelt es sich um ein Artefakt des Lösungsraums. Sie gibt an, aus welchen Implementierungskomponenten das Produkt zusammengesetzt werden muss. Wie auch im generativen Domänenmodell dargestellt, muss der Generator dazu über sogenanntes Konfigurationswissen verfügen. Dazu zählen Informationen darüber, wie die Komponenten kombiniert werden können, welche sich möglicherweise ausschließen und

welche Komponente eine andere bedingt (vgl. [Czarnecki/Eisenecker 2000, 6]). Darüber hinaus ist es bei der Spezifikation nicht zweckmäßig, jedes einzelne Detail angeben zu müssen. Im Falle des bereits angeführten Beispiels des Smartphones würde dies bedeuten, dass auch alle selbstverständlichen Eigenschaften mit angegeben werden müssten, also beispielsweise dass es über einen Akku, ein Display sowie eine Antenne verfügt. Die Spezifikation würde somit sehr schwierig werden, sodass es nur wenigen Benutzern gelingen dürfte, ein funktionsfähiges Produkt zu erhalten. Ein weiteres Problem besteht darin, dass zur vollständigen Beschreibung des Smartphones viele technische Details angegeben werden müssen, die dem Benutzer jedoch gar nicht vertraut sind. Denn die Implementierungskomponenten sind im Lösungsraum angesiedelt, der eine technische Sicht einnimmt. Daher verfügt der Generator über eine Reihe von Standardvorgaben, die verwendet werden, sofern der Benutzer nichts anderes spezifiziert. Auf diesem Wege kann sichergestellt werden, dass das gewünschte Produkt stets vollständig hergestellt wird (vgl. [Czarnecki/Eisenecker 2000, 132]).

Ein Generatorsystem ist im Gegensatz zu einem generativen System ausschließlich im Lösungsraum angesiedelt. Es besteht ebenfalls aus einem Generator, der Implementierungskomponenten zu Generaten zusammensetzt, verfügt aber weder über eine Spezifikation noch über eine DSL. Zur Entwicklung einer SPL kann im Lösungsraum daher ein Generatorsystem eingesetzt werden, ein Generatorsystem setzt aber weder ein generatives Domänenmodell noch SPLs voraus.

2.3 Entwicklungsprozess

2.3.1 Vorgehensweisen

Zur Entwicklung einer SPL stehen mehrere mögliche Vorgehensweisen zur Verfügung. Welche davon zur Anwendung kommt, hängt unter anderem davon ab, ob die Vielfalt in den entwickelten Produkten von Anfang an bekannt war oder ob möglicherweise schon bestehende Einzelsysteme in eine Produktlinie überführt werden müssen. Es können drei Möglichkeiten des Vorgehens unterschieden werden (vgl. [Acher 2011, 11]). Der proaktive Ansatz beginnt „auf der grünen Wiese“, das heißt es liegen keinerlei Altsysteme vor, die bei der Entwicklung berücksichtigt werden müssen. Jede Variabilität, die zukünftig benötigt werden könnte, soll von Beginn an berücksichtigt und implementiert werden. Dabei entsteht möglicherweise mehr Vielfalt als tatsächlich benötigt wird, die zu Mehraufwand und somit höheren Kosten führt.

Dieses Problem versucht der reaktive Ansatz zu umgehen: Es wird nur so viel Variabilität implementiert, wie zur Erstellung der gegenwärtig benötigten Produkte notwendig ist. Es besteht somit keine Gefahr, dass unnötige Funktionalitäten programmiert werden. Da sich Anforderungen häufig ändern können, ist es schwierig, die erforderliche Variabilität im Vorfeld korrekt einzuschätzen. Wird nun ein neues Produkt verlangt, kann es passieren, dass es sich an einer Stelle von allen bisherigen Varianten unterscheidet. Dies kann zum Beispiel nach sich ziehen, dass eine der bisherigen Komponenten aufgeteilt werden muss. Erweist sich

dabei die Architektur nicht als flexibel genug, kann dies umfangreiche Änderungen an der gesamten SPL nach sich ziehen. Dieser inkrementelle Ansatz ermöglicht somit einen schnellen und kostengünstigen Umstieg auf Software-Produktlinien, setzt jedoch eine generische Architektur voraus, die später entsprechend angepasst werden kann.

Häufig hat ein Unternehmen bereits einige Produkte in Form von Einzelsystemen entwickelt, die nachträglich in eine SPL überführt werden sollen. Dieses Szenario wird als extraktive Entwicklung bezeichnet. Dabei werden die bereits vorhandenen Produkte auf Unterschiede und Gemeinsamkeiten untersucht und darauf aufbauend eine Software-Produktlinie entwickelt (vgl. [Acher 2011, 11]).

Die gewählte Vorgehensweise wirkt sich maßgeblich auf die Größe und das Design der einzelnen Komponenten aus. Beim reaktiven Ansatz werden zunächst größere wiederverwendbare Bestandteile entwickelt, die später eventuell aufgeteilt werden müssen, während beim proaktiven Vorgehen einige entwickelten Bestandteile möglicherweise nie benötigt werden. Es ist also eine zentrale Entscheidung des Entwicklungsprozesses, welche Komponenten entworfen werden und wie die Anforderungen darauf verteilt werden. Die Kenntnis der jeweiligen Domäne ist also unverzichtbar, weshalb sich der folgende Abschnitt mit dem sogenannten *Domain Engineering* befasst.

2.3.2 Domain Engineering

Der Entwicklungsprozess einer SPL besteht, unabhängig von der gewählten Vorgehensweise, aus zwei Kernprozessen: Dem *Domain Engineering*, das als „Entwicklung *für* Wiederverwendung“ beschrieben werden kann, und dem *Application Engineering*, dass die „Entwicklung *durch* Wiederverwendung“ umfasst (vgl. [Acher 2011, 12]). Der erstgenannte Prozess kann wie folgt definiert werden:

> „ *Domain Engineering is the activity of collecting, organizing and storing past experiences in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable work products), as well as providing an adequate means for reusing these assets (i.e. retrieval, qualification, dissemination, adaption, assembly, and so on) when building new systems.*“
>
> [Czarnecki/Eisenecker 2000, 20]

Diese Definition ist eng an die Entwicklung des Quelltexts und der Erzeugung der Produkte angelehnt. Bei einer konventionellen Produktlinie im industriellen Bereich entspricht das Domain Engineering dem Bau der Fabrik, in welcher später die Herstellung erfolgt. Pohl et al. betonen hingegen die Bedeutung der Gemeinsamkeiten und Unterschiede, in dem sie Domain Engineering als „*the process […] in which the commonality and the variability of the product line are defined and realised* “ [Pohl et al. 2005, 21] definieren.

Beide Definitionen widersprechen sich nicht, betonen jedoch unterschiedliche Aspekte. Es ergibt sich ein Begriff des Domain Engineerings, der sich in drei Teilprozesse zerlegen lässt:

- Domänenanalyse

- Domänendesign
- Domänenimplementierung

Im vorhergehenden Abschnitt wurde bereits auf die Bedeutung der genauen Domänenanalyse hingewiesen. Sie stellt daher die erste Teilaktivität des Domain Engineerings dar. Sie umfasst unter anderem das *Domain Scoping*, in dem eine klare Eingrenzung der Domäne erfolgen soll. Dabei werden der Anwendungsbereich, involvierte Akteure und Ziele identifiziert. Dies ist notwendig, um die erforderliche Variabilität und die entsprechenden Abhängigkeiten analysieren zu können. Dabei wird gegebenenfalls auf bereits existierende Anwendungen zurückgegriffen, darüber hinaus aber auch auf Domänenexperten, Handbücher und Prototypen. Zum Abschluss der Domänenanalyse liegt ein konsistentes Domänenmodell vor. Es beinhaltet unter anderem ein Merkmalmodell, in dem die gefundene Variabilität sowie die Gemeinsamkeiten repräsentiert werden (vgl. [Czarnecki/Eisenecker 2000, 23ff]). In Abschnitt 3.3.3 wird diese Form der Variabilitätsmodellierung näher vorgestellt.

Im Anschluss an die Analyse erfolgt das Domänendesign. In diesem Teilprozess wird die Architektur der SPL festgelegt. Sie sollte hoch flexibel sein, da nicht ein Einzelsystem auf ihr basiert, sondern alle zu fertigenden Produkte (vgl. [Czarnecki/Eisenecker 2000, 28]). Die Einteilung der Software erfolgt auf der Grundlage der zuvor ermittelten Variabilität und den Gemeinsamkeiten der Domäne. Beides zusammen, flexible Architektur und passgenaue Komponenten, ermöglichen eine mit vielen Freiheitsgraden ausgestattete Konfiguration der Produkte (vgl. [Czarnecki/Eisenecker 2000, 9f]).

Die Domänenimplementierung umfasst die Programmierung der einzelnen Komponenten und weiterer Bestandteile. Das Ergebnis ist jedoch noch keine ausführbare Software, sondern nur eine Reihe von Einzelteilen. Diese werden dann im Application Engineering entweder manuell oder automatisiert zum gewünschten Produkt zusammengesetzt (vgl. [Czarnecki/Eisenecker 2000, 30]). Im zweiten Fall muss auch eine Software entwickelt werden, die diese Aufgabe übernimmt. Eine weitgehende Automatisierung wird durch die bereits vorgestellte generative Softwareentwicklung angestrebt.

2.3.3 Application Engineering

Während des Application Engineerings werden, basierend auf den Ergebnissen des Domain Engineerings, die einzelnen Produkte gefertigt (vgl. [Czarnecki/Eisenecker 2000, 30]). Je höher der Automatisierungsgrad ist, den die in der Domänenimplementierung entwickelte Fabrik erreicht, desto geringer ist der Aufwand zur Produktherstellung, wie Abbildung 2.2 veranschaulicht. Bei einer vollständigen Automatisierung reduziert sich der manuelle Aufwand, um ein Produkt herzustellen auf die Spezifikation mithilfe einer DSL durch den Benutzer. Aus der Spezifikation wird mithilfe des Konfigurationswissens des Generators die Konfiguration erstellt, die angibt, wie die einzelnen Implementierungskomponenten zusammengesetzt werden müssen. Anschließend erfolgt die Montage, an deren Ende ein fertiges Produkt zur Verfügung steht.

2.4 Merkmalorientierte Programmierung

Bei der merkmalorientierten Programmierung (engl. *Feature-oriented Programming, FOP*) handelt es sich um ein Programmierparadigma, dass insbesondere bei der SPL-Entwicklung Anwendung findet. Die bisher vorgestellten Methoden des Domain-Engineerings beschreiben, wie eine Software modelliert werden kann, die sich aus verschiedenen Merkmalen zusammensetzt. Merkmale sind also die Entitäten, welche die konzeptionelle Grundlage der Kapselung von Quelltext-Fragmenten bilden. Dies ist analog zur objektorientierten Programmierung, bei der die Klassen diese Rolle spielen. Die merkmalorientierte Programmierung bringt mit sich, dass Merkmale explizit herausgearbeitet werden müssen (vgl. [Cardozo et al. 2011, 1]). Insbesondere bei einer SPL, bei der die Variabilität in Form von Merkmalmodellen ausgedrückt wird, bringt dieses Vorgehen einige Vorteile mit sich. Dazu zählt unter anderem eine bessere Verständlichkeit des Quelltexts, da der Zusammenhang zwischen einem Merkmalmodell und der Implementierung eines Merkmals leichter ersichtlich ist. Weiterhin wird eine bessere Wartbarkeit erreicht, da bei Wegfall oder Änderung eines Merkmals die zu ändernden Stellen des Quelltexts leichter lokalisiert werden können. Da die Merkmale voneinander entkoppelt und somit weitestgehend unabhängig sind, funktionieren die nicht betroffenen Teile des Produkts weiterhin korrekt (vgl. [Batory et al. 2003, 1f]).

Generatorsystemen liegt allerdings meist keine merkmalorientierte Programmierung zugrunde. Stattdessen ist die Implementierung eines Merkmals auf den gesamten Quelltext verteilt. In diesem Fall wird von einer Delokalisierung des Merkmals gesprochen (vgl. [Bennicke/Rust 2004, 17]). Beispielsweise soll ein Produkt optional über Logging-Funktionalitäten verfügen. Eine mögliche Implementierungsform sieht so aus, dass jede einzelne Ausgabefunktion der Software um eine `IF`-Abfrage ergänzt wird, in welcher der entsprechende Parameter überprüft wird. Wird das entsprechende Merkmal ausgewählt, wird der entsprechende Code ebenfalls ausgeführt. Diese Variante ist in der Praxis häufig anzutreffen, auch bei Software-Produktlinien. Neben den angesprochenen Logging-Anweisungen sind auch Debug-Anweisungen häufig auf diese Weise implementiert. Dieser Ansatz hat jedoch einige Nachteile: Zum einen ist die Software schlechter wartbar, da bei der Änderung eines Merkmals viele unterschiedliche Code-Bestandteile angefasst werden müssen. Zum anderen kann der Fall eintreten, dass es dem Programmierer oder anderen Beteiligten nicht ersichtlich ist, dass

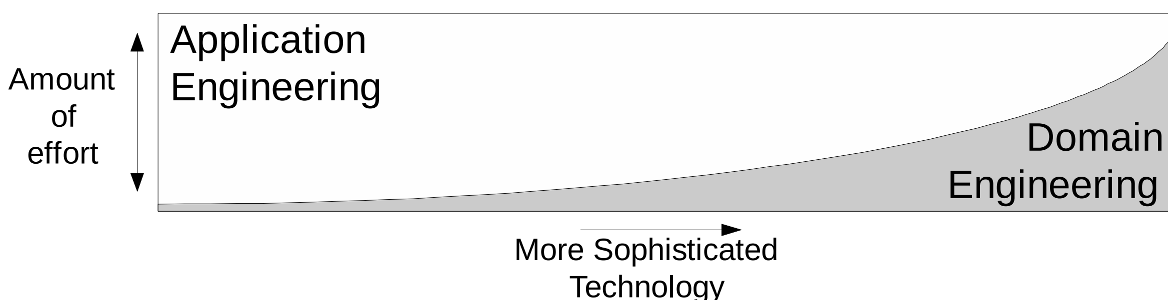

Abbildung 2.2: Zusammenhang zwischen investierten Aufwand in Domain und Application Engineering [Deelstra et al. 2005, 2]

es sich um ein Merkmal handelt, das global im System Verwendung findet (vgl. [Soloway et al. 1988, 1261]). Dies führt bei einem merkmalorientierten Vorgehen, beispielsweise in der Modellierung, zu Problemen, da jenes Merkmal nicht explizit in den Artefakten vorhanden ist.

2.5 Beziehungen zwischen Merkmalen

Bei der Entwicklung einer SPL wird angestrebt, dass die Merkmale unabhängig von anderen Merkmalen implementiert werden. Dies ist aber nicht immer uneingeschränkt möglich. Stattdessen liegen Abhängigkeiten zwischen den Merkmalen vor. Sie können sich zum einen direkt aus dem Problemraum der Domäne ergeben (vgl. [Czarnecki/Pietroszek 2006, 6]). In diesem Fall werden sie im Rahmen der Domänenanalyse erhoben und müssen modelliert werden, um sie bei der Implementierung berücksichtigen zu können. Beispielsweise können sich zwei Merkmale A und B gegenseitig ausschließen, es darf somit höchstens eines der beiden ausgewählt werden. Diese Einschränkung führt zu einer Verringerung der gültigen Merkmalkombinationen, da sowohl alle Spezifikationen als auch Konfigurationen, die sowohl A als auch B enthalten, ungültig sind.

Zum anderen ist es möglich, dass diese Beziehungen nur im Lösungsraum existieren. Dann handelt es sich um Abhängigkeiten zwischen Implementierungskomponenten. Beispielsweise kann in der Implementierung eines Merkmals ein eigener Typ definiert werden. Wird dieser auch noch an anderer Stelle verwendet, muss sichergestellt werden, dass die Typdefinition bekannt ist. Auf diese Weise entstehen Abhängigkeiten zwischen Merkmalen, die in der Domäne nicht enthalten sind (vgl. [Thaker et al. 2007, 1]).

Weiterhin soll eine Unterscheidung zwischen Constraints und Merkmalinteraktionen getroffen werden, auch wenn in der Literatur beide Begriffe teilweise synonym verwendet werden (vgl. [Ferber et al. 2002, 235]). Von Ersteren ist die Rede, wenn die Abhängigkeiten keinen Einfluss auf die Anzahl der gültigen Konfiguration hat. Beispielsweise derart, dass zusätzliche Anweisungen ausgeführt werden müssen, wenn eine bestimmte Merkmalkombination gewählt wurde. Dies ist zum Beispiel der Fall, wenn zwei Merkmalimplementierungen auf dieselbe Ressource zugreifen (vgl. [Weiss et al. 2005, 3]). Seien die beiden Merkmale mit A und B bezeichnet. Sofern nur ein Merkmal ausgewählt wurde, entsteht kein Problem. Wurde jedoch A und B selektiert, kann ein gleichzeitiger Schreibzugriff zu unerwünschten Verhalten führen. Daher muss das Produkt über einen zusätzlichen Mechanismus C verfügen, der den Zugriff auf die Ressource entsprechend regelt. Es handelt sich dabei lediglich um Hilfsanweisungen, die für das Funktionieren der Merkmale notwendig sind. Aus diesem Grund lässt sich argumentieren, dass dieser Zugriffsmechanismus nicht als eigenständiges Merkmal betrachtet werden muss, sondern als Bestandteil der Implementierung der beiden Merkmale. Es gibt jedoch kein allgemeingültiges Kriterium, wann eine Modellierung als eigenständiges Merkmal vorzuziehen ist und wann nicht. Wenn C auch unabhängig von den Merkmalen A und B selektiert werden kann, soll es in der vorliegenden Arbeit als Merkmal behandelt werden. Wenn C nicht separat gewählt werden kann, sondern sich stets aus A

und B ergeben muss, soll auf eine Modellierung von C verzichtet werden. Die Abhängigkeit muss aber dennoch dokumentiert werden.

Führen die Abhängigkeiten zwischen Merkmalen zu unerwünschten Nebeneffekten wird von *feature interference* gesprochen (vgl. [Filho/Redmiles 2007, 1]). Im Rahmen mehrerer Fallstudien stellte sie sich als eines der am häufigsten auftretenden Probleme bei der Umsetzung von SPLs heraus (vgl. [Deelstra et al. 2004, 175]). Durch die hohe Komplexität mit tausenden von Merkmalen sind die Abhängigkeitsverhältnisse nicht mehr unmittelbar überblickbar. Das Konfigurationswissen des Generators wird dadurch nur unzureichend gepflegt. In der Folge können Produkte generiert werden, die nicht funktionsfähig sind. Die Ursache liegt darin, dass Merkmalinteraktionen und Constraints oft unzureichend dokumentiert werden und deshalb auch nicht bei der Implementierung berücksichtigt werden. Da eine Vermeidung solcher Abhängigkeitsverhältnisse zwischen Merkmalen nicht immer möglich ist, müssen sie besser behandelt werden, um Probleme zu vermeiden. Im Abschnitt 3.3.4 wird daher gezeigt, welche Möglichkeiten Merkmalmodelle zur Darstellung von Interaktionen zur Verfügung stellen.

3 Variabilität

Zu Beginn dieser Arbeit wurde bereits das Konzept der Variabilität im Rahmen von SPLs kurz vorgestellt. In Rahmen dieses Kapitels soll nun eine ausführlichere Betrachtung erfolgen. Zuerst werden einige zentrale Begriffe definiert, bevor auf die unterschiedlichen Dimensionen des Variabilitätsbegriffs eingegangen wird. Anschließend wird in Abschnitt 3.3 auf die Modellierung der Variabilität eingegangen.

3.1 Definitionen

Kann ein Artefakt auf einfache Weise für einen bestimmten Anwendungsfall erweitert, angepasst oder konfiguriert werden, so verfügt es über Variabilität. Durch verschiedene Konfigurationen entstehen somit unterschiedliche Ausprägungsformen des Ausgangsgegenstandes. Diese werden als Variabilitätsobjekte bezeichnet, die veränderliche Eigenschaft als Merkmal (engl. *Feature*) (vgl. [Pohl et al. 2005, 59]). In der Literatur finden sich aber auch andere Definitionen, die von einem Merkmal im Sinne eines für den Endbenutzer sichtbaren Charakteristik des Systems sprechen. Im Rahmen dieser Arbeit wird jedoch die allgemeinere Variante verwendet, die sich allgemein auf *Stakeholder* bezieht (vgl. [Czarnecki/Eisenecker 2000, 38]). Durch die getroffene Verallgemeinerung können Merkmale flexibler eingesetzt werden, da sie auch nach außen nicht sichtbare Variabilität beschreiben können. Zudem ist es möglich, Merkmale hierarchisch anzuordnen, sodass Untermerkmale (engl. *sub-features*) entstehen (vgl. [Czarnecki et al. 2005a, 3]).

Damit von Variabilität gesprochen werden kann, müssen also mehrere Produkte vorhanden sein, sowie wie ein Ausgangsartefakt mit mindestens einem variablen Merkmal. Beispielsweise kann ein Softwaresystem für eine 32-Bit- oder 64-Bit-Architektur ausgeliefert werden. Der zugehörige Quelltext verfügt somit über das variable Merkmal „Architekturtyp“ und das für eine bestimmte Zielplattform kompilierte System stellt das konkrete Produkt dar. Würde die Software nur in einer 32-Bit-Version existieren, würde man nicht von Variabilität sprechen.

Als Variationspunkte, seltener auch Variabilitätspunkte, werden die Stellen eines Artefakts bezeichnet, an denen Variabilität auftreten kann (vgl. [Pohl et al. 2005, 61]). Abstrakt können sie auch als die Punkte eines Merkmals definiert werden, an dem es durch andere Merkmale erweitert werden kann. Die zur Verfügung stehenden Auswahlmöglichkeiten eines Merkmals werden als *Varianten* bezeichnet. Von Bedeutung ist, dass sich ein Variationspunkt auf mehrere Stellen und auf mehrere Artefakte verteilen kann, wie in Abbildung 3.1 illustriert.

Um erneut das Beispiel zweier Architekturen anzuführen: Das variable Merkmal ist weiterhin „Architekturtyp“. Die beiden Auswahlmöglichkeiten „32-Bit“ und „64-Bit“ stellen somit die beiden zulässigen Varianten dar. Im Generator existiert nun ein Variationspunkt, an dem zwischen beiden Möglichkeiten unterschieden wird, beispielsweise um andere Compiler-Optionen zu verwenden. Darüber hinaus wird aber auch in anderen Artefakten (Testfälle,

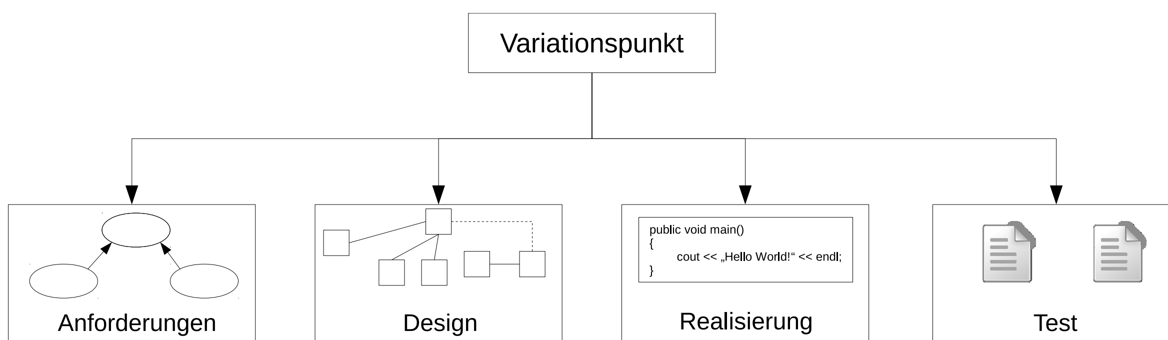

Abbildung 3.1: Variationspunkt (in Anlehnung an [Pohl et al. 2005, 83])

Klassenmodelle usw.) zwischen beiden Architekturen unterschieden. Die Variabilität befindet sich sowohl in den Anforderungsdokumenten, als auch in Klassenmodellen, im Quelltext und in den Testfällen. Die Auswirkungen auf alle diese Dokumente ergibt zusammen einen Variationspunkt für die Entscheidung zwischen 32-Bit- und 64-Bit-Architektur. Variabilität wirkt sich somit nicht nur auf Ebene des Quelltexts aus, sondern erstreckt sich auf den gesamten Entwicklungsprozess des Softwaresystems (vgl. [Czarnecki 2013]).

3.2 Dimensionen

Da der obige Variabilitätsbegriff noch recht allgemein ist, gibt es in der Literatur eine Reihe von Ansätzen, diesen zu verfeinern und zwischen verschiedenen Arten von Variabilität zu differenzieren. Diese sollen im folgenden als Dimensionen bezeichnet werden. So unterscheiden beispielsweise Pohl et al. zwischen einer zeitlichen und räumlichen Variabilität. Erstere liegt vor, wenn nur eine Instanz eines Artefaktes existiert, die sich im zeitlichen Verlauf jedoch verändert. Wird eine Software kontinuierlich weiterentwickelt, weist zum Beispiel der Quelltext eine zeitliche Variabilität auf. Dieses Verhalten wird unter den Begriffen *Versionierung* und *Software-Evolution* näher untersucht (vgl. [Madhavji et al. 2006, 8f]). Bei Software-Produktlinien steht die räumlichen Variabilität im Vordergrund. Diese tritt auf, wenn zum selben Zeitpunkt ein Artefakt in mehreren einander ähnlichen Ausprägungen vorliegt (vgl. [Pohl et al. 2005, 65f]).

Des Weiteren kann zwischen interner und externer Variabilität differenziert werden. Diese unterscheiden sich in der Sichtbarkeit gegenüber dem Kunden beziehungsweise Anwender. Kann dieser entscheiden, welches konkrete Variante eines Merkmals er nutzen möchte, handelt es sich um externe Variabilität (vgl. [Pohl et al. 2005, 68]). Das ist zum Beispiel der Fall, wenn der Kunde sich entscheiden muss, ob er entweder die 32-Bit- oder die 64-Bit-Version einer Software kauft. Es gibt jedoch auch Systeme, die nur in einer Ausführung existieren und dennoch beide Plattformen unterstützen. In diesem Fall spricht man von interner Variabilität, da die unterschiedliche Behandlung beider Architekturen dem Benutzer verborgen bleibt.

Zudem sollen noch die beiden Dimensionen statische und dynamische Variabilität eingeführt werden. Erstere beschreibt das Vorhandensein von sich nicht ändernden Artefakten, wie beispielsweise Quelltexte oder Modelle. Es sei darauf hingewiesen, dass sich diese im zeitli-

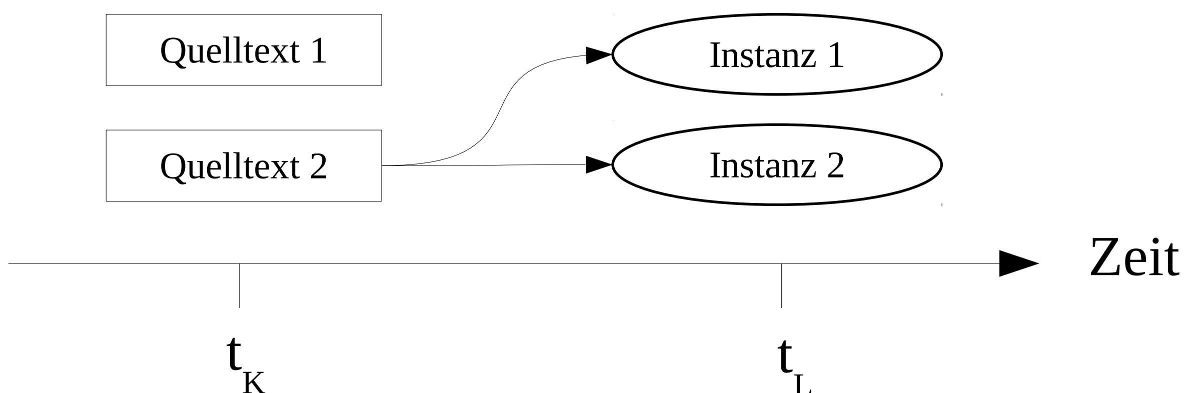

Abbildung 3.2: Zur Kompilierzeit t_K liegen statische Variabilitätsobjekte (rechteckig) vor, bei der Ausführung entstehen zur Laufzeit t_L mehrere dynamische Variabilitätsobjekte (rund)

chen Verlauf des Entwicklungsprozesses natürlich sehr wohl verändern können. Im Vergleich zu der Ausführung einer Software, bei der Änderungen im Sekundenbruchteil stattfinden, gelten diese Artefakte dennoch als statisch. Ist zur Kompilierzeit nur ein Quelltext vorhanden, kann dieser zu diesem Zeitpunkt über keine Variabilität verfügen. Zum einen durch die Kompilierung und zum anderen durch die Ausführung des übersetzten Programms, entstehen neue Artefakte. Diese können sich in verschiedenen Punkten voneinander unterscheiden und somit über Variabilität verfügen. Dabei sind vor allem unterschiedliche Verläufe des Kontrollflusses von Bedeutung, auf die im Abschnitt 5.5.2 eingegangen wird. Da diese Unterscheide nur zur Laufzeit auftreten, liegt dynamische Variabilität vor (vgl. [Bosch/Capilla 2012; Pohl et al. 2005, 20]). In Abbildung 3.2 ist der Zusammenhang beider Dimensionen illustriert.

Zusammenfassend lässt sich sagen, dass Variabilität in unterschiedlichen Formen auftritt, auch außerhalb von Software-Produktlinien. So verfügt auch herkömmliche Software sowohl über eine zeitliche als auch eine dynamische Variabilitätsdimension. Im Kontext von Software-Produktlinien und Generatorsystemen kommt vor allem der räumlichen Variabilität eine hohe Bedeutung zu. Für die vorliegende Arbeit ist zudem die Unterscheidung zwischen statischer und dynamischer Variabilität essentiell, auf sie wird unter anderen im Abschnitt 5.5.4 zurückgegriffen.

3.3 Variabilitätsmodellierung

3.3.1 Definition

Im vorangegangenen Abschnitt wurden die Basiskonzepte der Variabilität vorgestellt. Es wurden jedoch keine Aussagen darüber getroffen, wie diese dargestellt werden können. Möglich ist dies beispielsweise durch eine Summe aussagenlogischer Formeln (vgl. [She 2008, 6]). Bei über hundert Merkmalen trägt diese Variante jedoch wenig zum Verständnis bei, insbesondere bei der Kommunikation mit dem Endbenutzer. Aus diesem Grund wird Variabilität auf einem höheren Abstraktionsniveau in Form graphischer Modelle dargestellt, dies wird als Variabilitätsmodellierung bezeichnet. Sie leistet damit einen Beitrag zum Variabili-

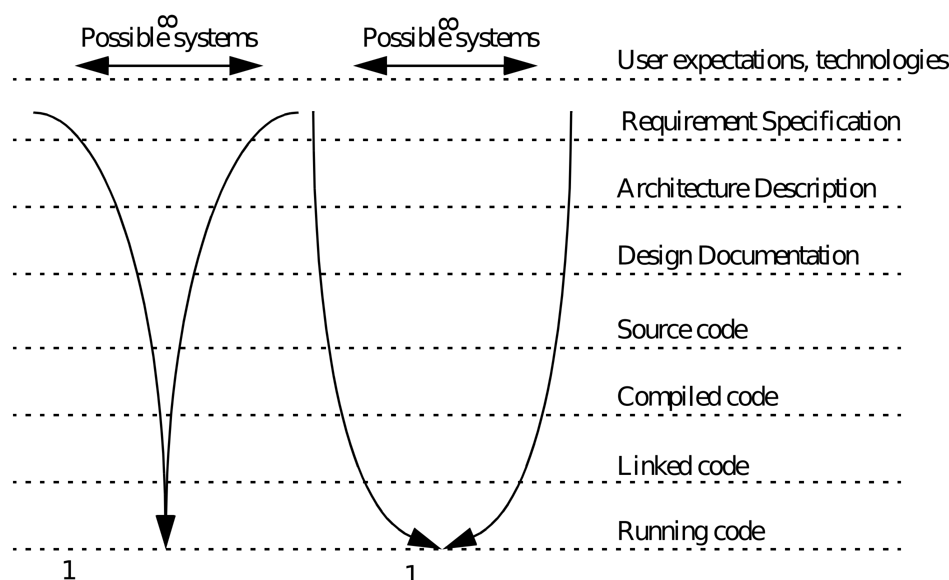

Abbildung 3.3: Variabilitätstrichter bei früher und später Variabilität [Gurp et al. 2001, 2]

tätsmanagement. Im Gegensatz zur herkömmlichen Modellierung können mehrere Produkte in einem Modell beschrieben werden (vgl. [Czarnecki et al. 2012, 1]).

3.3.2 Ansätze zur Variabilitätsmodellierung

Aufgrund der vielen Artefakte im Software-Entwicklungsprozess existiert auch eine Vielzahl von unterschiedlichen Modellierungstechniken und Modellen. So versucht beispielsweise die orthogonale Variabilitätsmodellierung sämtliche Variabilitätsinformationen in einem eigenen Modell darzustellen und so einen Querschnitt über alle Artefakte zu liefern. Dies führt dazu, dass einige Autoren kritisieren, dass orthogonale Modelle überladen wirken (vgl. [Pohl et al. 2005, 74f]). Andere Ansätze versuchen die bereits vorhandenen Modelle zu erweitern und mithilfe zusätzlicher Sprachelemente Variabilität auszudrücken. Des Weiteren wird zwischen Merkmal- und Entscheidungsmodellierung (engl. *Decision Modeling*) unterschieden (vgl. [Czarnecki et al. 2012, 1]). Merkmalmodellierung baut auf den bereits vorgestellten Konzept der obligatorischen und variablen Merkmale auf. Modelliert werden also die Gemeinsamkeiten und Unterschiede, die innerhalb einer Domäne auftreten. Als Notationsform werden in der Regel Merkmalmodelle gewählt, es existieren jedoch auch weitere Darstellungsformen (vgl. [Czarnecki et al. 2012, 2]). Die Entscheidungsmodellierung betrachtet stattdessen den Entwicklungsprozess. In dessen Verlauf werden Entscheidungen über die gewünschten Merkmale getroffen, welche die Anzahl der am Ende möglichen Produkte kontinuierlich einschränken. Dies ist in Abbildung 3.3 in Form eines sogenannten *Variabilitätstrichters* dargestellt. Zu Beginn der Entwicklung kann jedes beliebige System gebaut werden. Durch das Festlegen auf konkrete Anforderungen, dann auf eine Architektur, dann auf ein bestimmtes Design nimmt die Variabilität immer weiter ab, bis schließlich nach der erfolgten Konfiguration ein einzelnes Produkt übrig bleibt. In der Praxis werden beide Formen häufig eingesetzt, zudem sie sich nur in Details unterscheiden. Der einzige wesentliche Unterschied ist, dass bei

der Merkmalmodellierung sowohl Gemeinsamkeiten als auch variable Bestandteile berücksichtigt werden, während bei der Entscheidungsmodellierung nur Letztere betrachtet werden (vgl. [Czarnecki et al. 2012, 7]).

Einen gänzlich anderen Ansatz verfolgt die Common Variability Language (CVL). Sie definiert ein konkretes Produkt als Basismodell. In diesem werden nun die Variationspunkte gekennzeichnet und angegeben, welche Alternativen an dieser Stelle möglich sind. Es werden also nur die Unterschiede zwischen den Produkten modelliert. Zur graphischen Darstellung der Variationspunkte werden sogenannte *VSpecs* benutzt, die ähnlich zu den bereits angesprochenen Merkmalmodellen sind (vgl. [Czarnecki et al. 2012, 3ff]).

3.3.3 Merkmalmodelle

Im vorangegangenen Abschnitt wurden verschiedene Ansätze der Variabilitätsmodellierung vorgestellt. Bei der Entwicklung von Software-Produktlinien hat vor allem die Merkmalmodellierung eine hohe Verbreitung gefunden. Zur graphischen Darstellung existieren unterschiedliche Notationen. Beispielsweise existiert eine Erweiterung der Unified Modeling Language (UML) in Form sogenannter Variabilitätsdiagramme, die es erlaubt, Variationspunkte und Varianten zu modellieren (vgl. [Pohl et al. 2005, 78]).

Arten von Merkmalen

Im Zusammenhang mit Software-Produktlinien werden jedoch vor allem Merkmalmodelle eingesetzt. Dabei handelt es sich um gerichtete, azyklische Graphen (engl. *Directed Acyclic Graph*, *DAG*), mit genau einem Wurzelelement. Unter diesem werden die Merkmale und Untermerkmale hierarchisch angeordnet. Dabei wird einerseits zwischen abstrakten und konkreten Merkmalen sowie andererseits zwischen Einzelmerkmalen und Merkmalgruppen unterschieden. Abstrakte Merkmale besitzen im Gegensatz zu konkreten Merkmalen keine Implementierung, sie werden lediglich zur Gruppierung eingesetzt. Sofern allgemein von Merkmalen die Rede ist, sind im weiteren Verlauf dieser Arbeit stets konkrete Merkmale gemeint. Merkmalgruppen besitzen Untermerkmale, die wiederum auch Merkmalgruppen darstellen können. Im Falle der Smartphone-SPL handelt es sich bei „Architektur“ beispielsweise um eine abstrakte Merkmalgruppe, die beiden Auswahlmöglichkeiten stellen konkrete Einzelmerkmale dar. Eine Merkmalgruppe muss allerdings nicht zwangsläufig abstrakt sein. Damit ein Merkmal überhaupt ausgewählt werden kann, müssen auch alle entsprechenden Obermerkmale selektiert worden sein.

Jedes Merkmal kann nun mit einer Kardinalität in Form eines Intervalls $[a..b]$ versehen werden. In Falle von Einzelmerkmalen bedeutet dies, dass es mindestens a-mal aber höchstens b-mal ausgewählt werden darf. Gebräuchlich sind insbesondere die beiden Intervalle $[1..1]$ sowie $[0..1]$. Ersteres wird auch als obligatorisches Merkmal (engl. *mandatory feature*) bezeichnet, da es stets Bestandteil der Konfiguration sein muss. Bei Letzterem handelt es sich um ein optionales Merkmal (engl. *optional feature*), da es ausgewählt werden kann, aber

nicht muss. Bei einem Smartphone entspricht der Akku beispielsweise einem obligatorischem Merkmal. Eine Kamera stellt dagegen ein optionales Merkmal dar.

Die Kardinalität einer Merkmalgruppe gibt an, wie viele der zugehörigen Untermerkmale ausgewählt werden sollen. So bedeutet $[1..1]$, dass exakt ein Element der Gruppe in der Konfiguration enthalten sein muss, unabhängig von der Gesamtanzahl der Merkmale in der Gruppe. Dies entspricht einem exkluivem Oder (XOR), sodass von einer XOR-Gruppe gesprochen wird. Eine inklusives Oder (OR) liegt hingegen vor, wenn die einzige Bedingung ist, dass mindestens ein Merkmal ausgewählt werden muss. Besteht eine Merkmalgruppe aus drei Untermerkmalen, so kennzeichnet das Intervall $[1..3]$ eine OR-Gruppe. Die Bezeichnungen sind an die Operatoren der Aussagenlogik angelehnt, da sie nach dem selben Prinzip funktionieren. Soll die Smartphone-Software entweder auf 32-Bit- oder auf 64-Bit-Architekturen ausführbar sein, so handelt es sich bei der Architektur um eine XOR-Gruppe. Wenn hingegen auch Produkte generiert werden können, die auf beiden Architekturen lauffähig sind, handelt es sich um eine OR-Gruppe.

Cross-tree-Constraints erlauben die Angabe expliziter Abhängigkeiten zwischen zwei oder mehr Merkmalen (vgl. [Thaker et al. 2007, 1]). Die Merkmale, für die solche Constraints definiert werden, müssen weder auf der selben hierarchischen Ebene angeordnet sein, noch Bestandteil desselben Zweigs sein. Sie dienen daher zur Abbildung von Merkmalinteraktionen und können sowohl auf Ebene der Merkmale als auch der Variationspunkte modelliert werden. Merkmalmodelle verfügen über zwei verschiedene Coress-tree-Constraints, die *requires* und *excludes* genannt werden (vgl. [Pohl et al. 2005, 79f]). Erstere drückt aus, dass ein Merkmal ein anderes bedingt. Wenn ein Merkmal A selektiert wurde und eine Constraint $A\ requires\ B$ existiert, muss eine gültige Konfiguration ebenso Merkmal B enthalten. Es handelt sich um eine gerichtete Verbindung von A nach B. Es ist daher zulässig nur Merkmal B auszuwählen. Die Angabe $A\ excludes\ B$ hingegen bedeutet, dass beide Merkmale nicht zusammen selektiert werden können.

Graphische Darstellung

Merkmalmodelle stellen die zuvor beschriebenen Merkmale und Kardinalitäten graphisch dar. Es haben sich drei verschiedene Darstellungsformen etabliert. Die ursprüngliche Version ist in Abbildung 3.4 dargestellt. Es erfolgt keine Angabe der Kardinalitäten, stattdessen wird eine graphische Notation verwendet. Es gibt vier graphische Gestaltungselemente, die vier konkrete Intervalle repräsentieren.

Später wurden von Czarnecki et al. [2005] Merkmalmodelle vorgeschlagen, in denen die Kardinalitäten direkt angegeben werden. Sie sind identisch aufgebaut, es werden jedoch keine graphischen Elemente zur Kennzeichnung verwendet. Das eben vorgestellte Modell ist in Abbildung 3.5 mithilfe dieser Notation noch einmal abgebildet. Beide Diagramme sind semantisch äquivalent. Durch die Angabe von Kardinalitäten wird eine höhere Flexibilität erreicht, da nun auch Merkmalgruppen ermöglicht werden, aus denen zum Beispiel mindestens zwei aber höchstens drei Elemente ausgewählt werden dürfen.

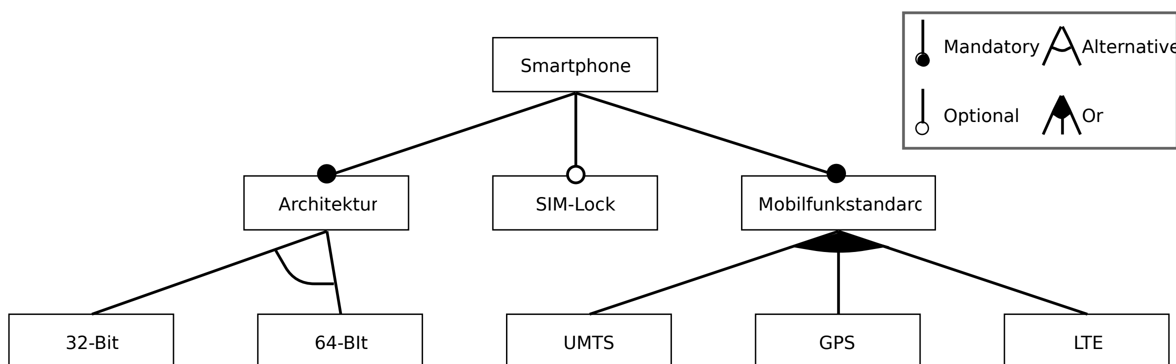

Abbildung 3.4: Ein einfaches Merkmalmodell mit Legende

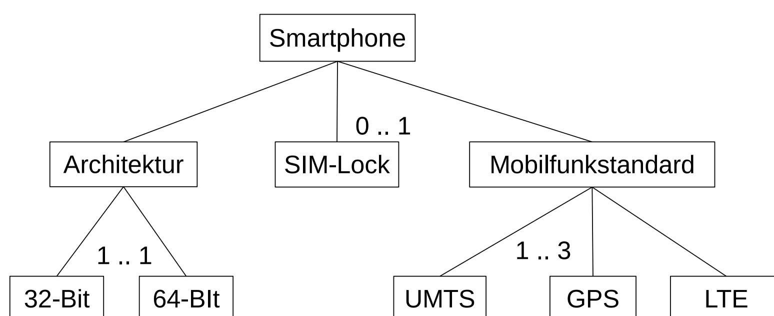

Abbildung 3.5: Kardinalitätsbasiertes Merkmalmodell

Eine weitere Ergänzung stellen wahrscheinlichkeitsbasierte Merkmalmodelle dar. Sie ergänzen das Diagramm um zusätzliche Informationen in Form bedingter Wahrscheinlichkeiten. Zum einen kann somit die Wahrscheinlichkeitsverteilung innerhalb einer Merkmalgruppe angegeben werden. Zum anderen können auch Cross-tree-Angaben erfolgen, also wie wahrscheinlich es ist, dass Merkmal a selektiert wird, wenn bereits bekannt ist, dass Merkmal b in der Auswahl enthalten ist. Diese Informationen werden auch als „soft constraints“ bezeichnet, da sie Präferenzen ausdrücken, dabei aber nicht so streng sind wie eine requires- beziehungsweise excludes-Beziehung. Diese können als Spezialisierung aufgefasst werden. Eine bedingte Wahrscheinlichkeit von 0% entspricht einer excludes- und 100% entsprechend eine requires-Beziehung (vgl. [Czarnecki et al. 2008, 1]). Die notwendigen Angaben können dabei aus bisherigen Konfigurationen gewonnen werden. Zu diesem Zweck werden die einzelnen Merkmale auf stochastische Unabhängigkeit geprüft. Kann diese nicht bestätigt werden, wird der entsprechende Wert in das Diagramm übernommen (vgl. [Czarnecki et al. 2008, 6ff]). Auf diese Weise können die Angaben inkrementell verbessert werden, da jede neu erstellte Konfiguration Auswirkungen auf das Merkmalmodell hat. Weiterhin können jedoch auch andere Daten die Grundlage der bedingten Wahrscheinlichkeiten bilden. Ist beispielsweise aus empirischen Marktforschungsstudien bekannt, dass bestimmte Kombinationen von Merkmalen besonders beliebt sind, kann dies ebenso in das Merkmalmodell aufgenommen werden. Es stellt sich jedoch die Frage nach dem Mehrwert der bedingten Wahrscheinlichkeiten. Wie bereits erwähnt, haben sie keinen Einfluss auf die Anzahl der gültigen Konfiguration, erlauben aber das Identifizieren von häufiger auftretenden Merkmalkombinationen. Dies er-

laubt es, den Konfigurationsprozess dynamischer zu gestalten, indem Standardwerte definiert werden. Wenn die Merkmale a und b in 90% der Fälle zusammen ausgewählt werden, so kann automatisch b selektiert werden, wenn zuvor a ausgewählt wurde. Czarnecki et al. geben an, dass ab einer bedingten Wahrscheinlichkeit von 80% eine solche Voreinstellung sinnvoll ist, da sie die Komplexität für den Benutzer reduziert und nur in einigen Fällen nachgebessert werden muss (vgl. [Czarnecki et al. 2008, 3]). Herkömmliche Merkmalmodelle erlauben nur statische Standardwerte, die nicht während der Konfiguration auf die Entscheidungen des Benutzers reagieren können.

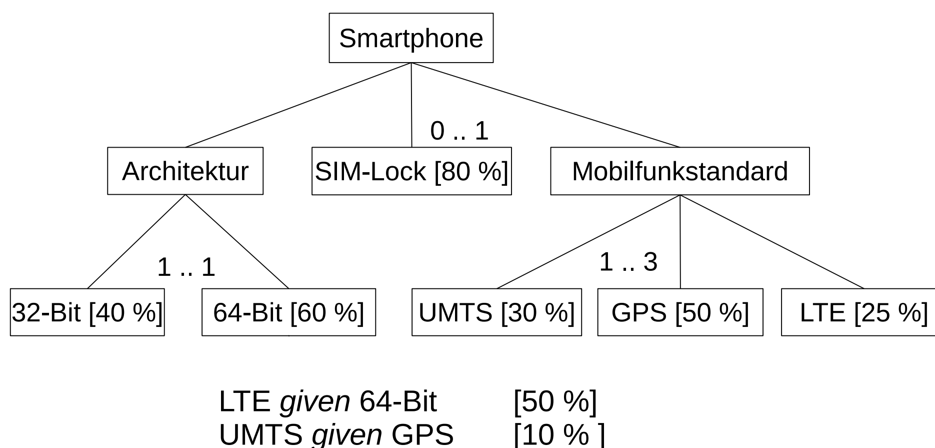

Abbildung 3.6: Wahrscheinlichkeitsbasiertes Merkmalmodell

3.3.4 Constraints-Modellierung

Im vorangegangenen Abschnitt wurden die Möglichkeiten von Merkmalmodellen zur Darstellung von Constraints vorgestellt. Obwohl sich die zur Verfügung stehenden requires- und excludes-Beziehungen in der Praxis als nicht ausreichend erwiesen haben, gibt es keine direkte Erweiterung von Merkmalmodellen, die komplexere Constraints oder Merkmalinteraktion erlauben (vgl. [Filho/Redmiles 2007, 2]). Auch in einer Reihe anderer Modellierungsansätze steht keine bessere Notation zur Verfügung (vgl. [Ferber et al. 2002, 250]). Nichtsdestoweniger gibt es Notationen, die auch Merkmalinteraktionen berücksichtigen. Dabei handelt es sich jedoch nicht um kompatible Erweiterungen von Merkmalmodellen, sondern um eigenständige Verfahren, wie sie unter anderem bei Kim et al. [2008] zu finden sind.

Die Modellierung von Merkmalbeziehungen ist im Falle der vorliegenden ADS-Makros allerdings von Bedeutung, da diese weder als SPL konzipiert wurden, noch eine merkmalorientierte Programmierung erfolgte. Zudem stellte ADS mit Entscheidungstabellen ein Konstrukt bereit, mit dem auf einfache Weise umfangreiche Merkmalinteraktionen implementiert werden können. Es kann daher zumindest vermutet werden, dass bei der Variabilitätsextraktion vielfältige Beziehungen zwischen Merkmalen gefunden werden. Um sicherzustellen, dass diese angemessen modelliert werden können, wird in dieser Arbeit eine rein mathematische statt einer graphischen Notation verwendet.

Um diese mathematische Form zu erklären, soll sie zuerst anhand der beiden Constraints erklärt werden, die von Merkmalmodellen unterstützt werden. Beide Varianten lassen sich auch mithilfe der booleschen Algebra darstellen. Dabei entspricht eine requires-Abhängigkeit der logischen Implikation, für die der Operator $\Rightarrow$verwendet wird. Die Formel $A \Rightarrow B$ ist somit äquivalent zur eben beschriebenen requires-Beziehung. Durch die Negation kann ausgedrückt werden, dass ein Merkmal nicht ausgewählt werden darf: $\neg A$ Der Ausdruck $A \Rightarrow \neg B$ entspricht somit der excludes-Beziehung.

Auch die im vorangegangenen Abschnitt beschriebenen Variabilitätsmechanismen lassen sich durch Constraints ausdrücken. Beispielsweise ließe sich ein Merkmal C mit einem optionalen Untermerkmal D wie folgt darstellen: $D \Rightarrow C$. Diese Constraints ergeben sich, im Gegensatz zu den Cross-tree-Constraints, aus der hierarchischen Anordnung der Merkmale im Merkmalmodell. Es wird deutlich, dass sich auf diese Weise das komplette Merkmalmodell durch eine Menge aussagenlogischer Formeln abbilden lässt. Diese wird auch als Wissensbasis bezeichnet. Für eine gültige Konfiguration muss jede einzelne Formel zu wahr evaluieren.

Mit den Möglichkeiten der booleschen Algebra können nun auch komplexe Merkmalinteraktionen ausgedrückt werden: Beispielweise gibt es Smartphones, die mehrere Mobilfunkstandards, wie UMTS, LTE und GPRS parallel unterstützen können. Die drei Varianten verwenden unterschiedliche Frequenzen zur Kommunikation. Wenn das Smartphone mehrere Mobilfunkstandards unterstützt, ist es somit notwendig, dass die Sende- und Empfangsfrequenz veränderbar ist. Diese Funktionalität ist nicht notwendig, wenn nur ein Merkmal selektiert wurde. Für den Endbenutzer, der sich sein persönliches Produkt konfiguriert, ist diese Information auch nicht relevant. Es handelt sich um interne Variabilität. Für die Produktion des Smartphones ist es durchaus wichtig zu wissen, dass diese zusätzliche Funktionalität vorhanden sein muss. Mit den bisher vorgestellten Möglichkeiten, kann diese Beziehung nicht ausgedrückt werden. Sie kann jedoch durch folgende Formel modelliert werden:

$(LTE \wedge UMTS) \vee (LTE \wedge GPRS) \vee (UMTS \wedge GPRS) \Rightarrow Frequenzwechsler$

4 Application Development System

In Kapitel 2 wurden die Grundlagen von Software-Produktlinien und Generatoren erläutert. Mit ADS wird nun eine Programmiersprache vorgestellt, mit der sich derartige generative Systeme realisieren lassen. Zunächst wird im folgenden Abschnitt der allgemeine Aufbau sowie der Ablauf des Generierungsprozesses beschrieben. Anschließend werden einige relevante Eigenschaften der Sprache im Detail vorgestellt. Dazu zählen Kontrollfluss- und Wiederholungsanweisungen sowie Variablen und Parametern. Die bestehenden Makros wurden zwar nicht als SPL entworfen, sollen im Rahmen von AmAVaG aber in ein generatives Domänenmodell überführt werden. In Abschnitt 4.3 wird deshalb gezeigt, wie Konfigurationswissen in ADS implementiert werden kann.

4.1 Aufbau und Ablauf

ADS stellt umfangreiche Programmbibliotheken zur Entwicklung von Anwendungsprogrammen zur Verfügung. Die Produkte werden in Textform erzeugt und können daher in jeder textbasierten Zielsprache erstellt werden. ADS wird jedoch vor allem zur Generierung von Anwendungsprogrammen in Programming Language One (PL/I) sowie der Common Business Oriented Language (COBOL) eingesetzt. Es setzt sich aus mehreren Bestandteilen zusammen, zu nennen sind insbesondere Prozessoren, Microcode und Makros. ADS selbst ist in der Programmiersprache Processor Development Language (PDL) implementiert. Bei deren Kompilierung wird sogenannter ausführbarer *Microcode* erzeugt (vgl. [DSTG 2012a, 16]). Zudem stellt ADS eine Reihe von Generatoren in Form von Microcode zur Verfügung, die als Prozessoren bezeichnet werden. Bei Makros handelt es sich um den eigentlichen Quelltext. Sie bestehen aus ADS-Anweisungen, Eingabewerten für den jeweiligen Prozessor und COBOL-Code. Makros können dabei als herkömmliche Funktionen betrachtet werden, da sie über Ein- und Ausgabeparameter verfügen und sich gegenseitig aufrufen können.

Der Generierungsvorgang sieht wie folgt aus: Der Prozessor liest ein Makro ein und verarbeitet die darin enthaltenen ADS-Anweisungen. Dabei können drei unterschiedliche Arten von Dateien erzeugt werden: *OUT-Files*, *PRINT-Files* und *DOC-Files*. Erstere stellen den ausführbaren Teil der Produkte dar und werden durch die Ausführung der Makros erzeugt. Die OUT-Files enthalten also ausschließlich Code der Zielsprache, der anschließend kompiliert und ausgeführt werden kann, sofern gültiger Quelltext generiert wurde. Die beiden anderen Ausgabearten bilden zusammen die Dokumentation: Bei PRINT-Files handelt es sich um Logdateien, die den Generierungsdurchlauf protokollieren. Diese Dateien werden bei jedem Generatordurchlauf erzeugt. Sie können unter anderem Baum- und Einrückdiagramme, Belegungstabellen oder Struktogramme enthalten. Die DOC-Sätze hingegen enthalten weitere Angaben zur Verwendung einzelner Programmbestandteile, wie Felder und Subroutinen (vgl. [DSTG 2012a, 15ff]).

Eine Eigenschaft von ADS ist, dass Zielsprache und Implementierungssprache des Generators unterschiedlich sind. Bei COBOL handelt es sich um eine Programmiersprache, die nicht direkt interpretiert werden kann, sondern zuvor kompiliert werden muss. Wird ein Produkt erzeugt, das nicht übersetzt werden kann, führt das während der Generierung nicht zwangsläufig zu einem Fehler. Es ist somit von hoher Bedeutung, dass der Generator über ein umfangreiches Konfigurationswissen verfügt, um solche Fälle zu vermeiden.

4.2 Sprachkonstrukte

ADS ist eine Turing-vollständige Sprache und verfügt somit über Kontrollflussstrukturen, Wiederholungsanweisungen und Variablen. Es gibt jedoch weder Klassen noch Typen. Darüber hinaus verfügt ADS über sprachspezifische Konstrukte, die in vielen anderen Programmiersprachen nicht anzutreffen sind. Dazu gehört zum Beispiel ein Baustein-Konzept. Dieses erlaubt es jederzeit, an verschiedenen Stellen der OUTFiles Text einzufügen. Da die vorliegenden Makros davon jedoch keinen Gebrauch machen, soll es nicht näher betrachtet werden. Stattdessen sollen im Folgenden die genannten grundlegenden Elemente der Sprache und ihre jeweilige Syntax vorgestellt werden, da ein gutes Verständnis der Sprache vonnöten ist, um anschließend Variabialitätsinformationen extrahieren zu können.

4.2.1 Kontrollflussstrukturen

ADS verfügt über `IF`-Anweisungen mit einem optionalen `ELSE`-Zweig. Diese weisen jedoch eine erhebliche Einschränkung auf: Es kann nur eine einzelne Bedingung überprüft werden. Es ist beispielsweise möglich zwei Variablen `A` und `B` auf Gleichheit zu testen mit folgendem Ausdruck: `#B.EQ.#A` Nicht möglich ist es allerdings zu überprüfen, ob sowohl `#A.EQ.1` als auch `#B.EQ.2` erfüllt ist. Um dies zu erreichen müssen entweder zwei `IF`-Abfragen geschachtelt oder sogenannte Entscheidungstabellen verwendet werden. Dabei handelt es sich um ein Sprachmittel, bei dem die Ausführung von sogenannten Aktionen von einer Reihe von Bedingungen abhängig gemacht werden kann. Eine Aktion stellt dabei eine Gruppe von Anweisungen dar, die gemeinsam ausgeführt werden. Dies ist vergleichbar mit einem zutreffenden Fall einer `switch`-Anweisung. Entscheidungstabellen sind jedoch deutlich flexibler. Bei einem `switch` kann immer nur ein Treffer erfolgen, es werden aber gegebenenfalls auch die folgenden `case`-Blöcke ausgeführt. Entscheidungstabellen hingegen ermöglichen eine detaillierte $m : n$-Zuordnung zwischen Aktionen und Regeln. Letztere bestehen aus Bedingungs- und Aktionsanzeigern. Es können beliebig viele Bedingungen definiert werden, die entweder zu $true$ oder $false$ evaluieren. Die Bedingungsanzeiger geben an, welche Wahrheitswert die Bedingungen annehmen müssen, damit die Regel als erfüllt gilt. Es kann zwischen Y für $true$ und N für $false$ gewählt werden. Zudem steht ein „`-`“ zur Verfügung, das anzeigt, dass die Bedingung für diese Regel irrelevant ist. Listing 4.1 zeigt eine einfache Entscheidungstabelle mit drei Bedingungen sowie je zwei Regeln und Aktionen. Tabelle 4.1 stellt eine alternative Darstellung dar. Damit die Regel 1 der Entscheidungstabelle zutrifft, muss folgendes erfüllt

```
1  .DTSTART
2  .DT-01.CO.CONSTANTS,    YY
3  .DT-02.CO.DEFAULTS,     NY
4  .DT-03.CO.NUMBER,       -Y
5  .DT                     XX,
6  .*Aktion 1
7  .DT                     -X,
8  .*Aktion 2
9  .DTEND
```

Listing 4.1: Implementierung einer Entscheidungstabelle

	1	2
Bedingung 1	Y	N
Bedingung 2	Y	N
Aktion 1	X	X
Aktion 2	-	X

Tabelle 4.1: Entscheidungstabelle aus Listing 4.1 in Tabellenform

sein: Variable `01` muss `CONSTANTS` enthalten. Variable `02` darf `DEFAULTS` *nicht* enthalten, da der Bedingungsanzeiger N als Negation der Bedingung wirkt. Der Wert der Variable `03` hat keinen Einfluss, da die Bedingung keinen Einfluss auf die Erfüllung der Regel hat.

Aktionsanzeiger verknüpfen die Aktionen mit den Regeln. Sie geben an, welche Aktionen ausgeführt werden, wenn eine bestimmte Regel erfüllt ist. Für jede Aktion wird ein eigener Aktionsanzeiger definiert. Bei einem Aufruf einer Entscheidungstabelle kann jede Aktion aber höchstens einmal ausgeführt werden, selbst wenn mehrere passende Regeln erfüllt sein sollten.

Der Aktionsanzeiger XX gibt an, dass die erste Aktion bei jeder erfüllten Regel aktiv wird. Die zweite Aktion wird jedoch nur durch die Erfüllung der zweiten Regel ausgeführt. Weiterhin ist zu erwähnen, dass Entscheidungstabellen nicht geschachtelt werden können und keine identischen Regeln existieren dürfen (vgl. [DSTG 2012a, 167f]).

4.2.2 Wiederholungsanweisungen

ADS kennt vier Varianten von Wiederholungsanweisungen: Es gibt sowohl kopf- als auch fußgesteuerte Schleifen, die auch als `WHILE`- und `UNTIL`-Schleifen bezeichnet werden. Weiterhin existieren Zählschleifen, die sich auch mit den beiden eben genannten Varianten kombinieren lassen. Zudem verfügt ADS über `FOREVER`-Schleifen, allgemein als Endlosschleifen bekannt, die durch einen expliziten Abbruch-Befehl wie `QUIT` oder `SKIP` verlassen werden müssen (vgl. [DSTG 2012b, 31ff]).

```
1 .DO-<pb>, <anzahl> [, <kommentar>]
2 .*Anweisungen
3 .DOEND-<pb>
```

Listing 4.2: Schleife

```
1 .*Positionsparameter
2 .ADD FDOCD, <parameter1>, <parameter2>
3 .*Schlüsselwortparameter
4 .PROG-LIST1, AUTHOR <parameterwert1>, <parameter2>
```

Listing 4.3: Verwendung von Parametern

Diese Vielfalt an Wiederholungsanweisungen ist zwar ungewöhnlich, allerdings dient es lediglich dazu, dem Entwickler Arbeit abzunehmen. Auch mit den beispielsweise aus C++ bekannten Schleifen lässt sich ein identisches Verhalten herstellen, es steht nur kein explizites Konstrukt für jeden Anwendungsfall zur Verfügung. Listing 4.2 zeigt eine Implementierung einer `WHILE`-Schleife. Im Schleifenkopf werden der Name der Schleife und in diesem Fall eine feste Anzahl an Wiederholungen angegeben. Es folgt der Rumpf, der beliebigen Anwendungscode enthalten kann. Der Befehl `DOEND-<pb>` beendet schließlich die Schleife.

4.2.3 Variablen und Parameter

Zunächst soll zur besseren Verständlichkeit eine Unterscheidung zwischen den Begriffen *Parameter* und *Variable* getroffen werden. Erster wird im Sinne eines Übergabeparameters verwendet, bezeichnet also die Werte, mit denen ein Makro aufgerufen wird. Ein Parameter ist für die Dauer der Ausführung des Makros konstant. Variablen hingegen werden innerhalb eines Makros verwendet und ihr Wert kann sich jederzeit ändern. In ADS fehlt diese Unterscheidung jedoch vollständig. Für jeden Parameter steht eine entsprechende Variable zur Verfügung, auf die innerhalb des Makros auch schreibend zugegriffen werden kann. Im ADS-Handbuch wird ausschließlich von Parametern und nie von Variablen gesprochen. Das entspricht jedoch nicht der mathematischen Bedeutung der Begriffe. Im weiteren Verlauf dieser Arbeit wird daher zwischen Variablen und Parametern unterschieden, unabhängig von der Formulierung in ADS. Parameter repräsentieren somit stets Werte, mit denen das Makro aufgerufen wurde, auch wenn diese zur Laufzeit verändert worden sind.

In ADS werden die Parameter unterschieden in Positions- und Schlüsselwortparameter. Beide Varianten sind in Listing 4.3 dargestellt. Bei Ersteren erfolgt die Verknüpfung des Wertes anhand der Position in der Parameterliste und bei Letzteren durch ein eindeutiges Schlüsselwort (vgl. [DSTG 2012a, 148]).

Parameter können weiterhin bezüglich ihres Gültigkeitsbereichs in globale und lokale Parameter unterteilt werden. Erstgenannte sind nicht nur innerhalb eines Makros gültig sondern innerhalb des gesamten Systems, das sich aus verschiedenen Makros und Prozessoren zusammensetzen kann. Sie werden meist zur Steuerung des Ablaufs der Generierung genutzt und

```
1 .SET-A=Eins
2 .DEF-A=Zwei
```

Listing 4.4: Schreibzugriffe auf Variablen

enthalten zum Beispiel Angaben zum Format des Programms, über den Umfang der Generierungsliste oder können den Syntax der Makros beeinflussen (vgl. [DSTG 2012a, 111ff]). Dem gegenüber stehen lokale Parameter, die nur innerhalb eines einzelnen Makros Gültigkeit besitzen. In diese Kategorie fallen die Aufrufparameter eines Makros. Diese werden mit `<01> .. <99>` bezeichnet. Der Parameter `<00>` kann nicht gesetzt werden, denn er enthält stets die Anzahl der übergebenen Parameter (vgl. [DSTG 2012a, 159]).

Variablen verfügen ebenfalls nur über einen lokalen Gültigkeitsbereich. Es gibt zusätzlich sogenannte statische Variablen, die ihren Wert behalten, auch wenn das Makro bereits verlassen wurde. Sie werden eingesetzt, um bei einem mehrmaligen Aufruf auf frühere Variablenwerte zugreifen zu können.

Eine explizite Wertzuweisung an Variablen erfolgt durch bestimmte Makroanweisungen, beispielsweise durch die Befehle `DEF` oder `SET`. Die Syntax ist in Listing 4.4 gezeigt. Bei der zweiten Variante handelt es sich um eine sogenannte *bedingte Wertzuweisung*. Wurde der Variablen `B` zuvor ein von *Nullstring* verschiedener Wert zugewiesen, erfolgt keine Änderung. Dies ist zu beachten, wenn aus ADS-Code Datenflussgraphen erzeugt werden, die in Abschnitt 5.5.3 noch vorgestellt werden. Darüber hinaus gibt es eine Vielzahl weiterer Möglichkeiten der Wertzuweisung, die an dieser Stelle nicht explizit vorgestellt werden sollen (vgl. [DSTG 2012a, 159ff]).

Es sei noch angemerkt, dass es in ADS nur einen Sichtbarkeitsbereich gibt. Variablen und Parameter können daher nicht verdeckt werden, sondern sind stets innerhalb des gesamten Makros sichtbar. Weiterhin verfügt ADS nicht über ein Typsystem, es existieren daher auch keine booleschen Variablen. Stattdessen können ihnen aber die Werte `0` beziehungsweise `1` zugewiesen werden. Weiterhin gibt es keine logischen Operatoren, wie beispielsweise `&&` oder `||`. Sollen Variablen miteinander verknüpft werden, muss daher entweder auf arithmetische Operationen oder Entscheidungstabellen zurückgegriffen werden. Zudem existieren einige Einschränkungen bezüglich der Variablennamen, die an dieser Stelle jedoch nicht im Detail vorgestellt werden sollen. Aus diesem Grund werden in den Listings dieser Arbeit meist Variablennamen wie `01` oder `02` verwendet, auch wenn diese nicht sprechend sind. Dieser Umstand ist für die Benennung von Merkmalen relevant, auf die in Abschnitt 5.9 eingegangen wird.

4.3 Implementierung von Konfigurationswissen

In diesem Abschnitt sollen mögliche Implementierungsformen von Sachverhalten gezeigt werden, die häufig im Kontext von SPLs auftreten. Dies dient zum einen dazu zu zeigen, dass diese mithilfe von ADS entwickelt werden können. Zum anderen ist dies der Ausgangspunkt

```
1 .IF.A.EQ.1
2 .IF.B.EQ.1
3 .* Anweisungen
4 .IFEND
5 .IFEND
```

Listing 4.5: Merkmalinteraktionen mittels geschachtelter `IF`-Abfragen

```
1 .DTSTART
2 .DT-A.EQ.1,      Y
3 .DT-B.EQ.1,      Y
4 .DT              X,
5 .*Aktion 1
6 .DTEND
```

Listing 4.6: Merkmalinteraktionen mittels Entscheidungstabelle

der Variabilitätsextraktion, auf die im nächsten Kapitel eingegangen wird. Es gibt fast immer mehrere Möglichkeiten der Implementierung von Konfigurationswissen, die semantisch äquivalent sind. Derzeit liegen nur wenig Erkenntnisse darüber vor, welche dieser verschiedenen Möglichkeiten in der Praxis vorzugsweise benutzt werden und in welchem Umfang die Makros überhaupt über Konfigurationswissen verfügen. Die nachfolgenden Quelltextbeispiele stellen somit vom Autor der vorliegenden Arbeit gefundene Lösungen dar. Sie geben zugleich den Rahmen der Variabilitätsextraktion dar, da diese anhand der in diesem Abschnitt gezeigten Beispiele erfolgt.

Soll eine exklusive Auswahl implementiert werden, kann hierzu ein `IF` oder `SELECT` verwendet werden. Letzteres entspricht einer `switch`-Anweisung, wie sie beispielsweise in C++ oder Java zur Verfügung steht. Sofern nur eine Bedingung überprüft werden muss, erfolgt die Implementierung wie in Abschnitt 4.2.1 gezeigt.

Hängt die Ausführung von Anweisungen von mehr als einer Bedingung ab, gibt es zwei Möglichkeiten. Listing 4.5 zeigt eine verschachtelte `IF`-Struktur. Zeigen die Variablen A und B die Auswahl zweier Merkmale an, so kann auf diese Weise dafür gesorgt werden, dass Anweisung nur dann ausgeführt werden, wenn beide Merkmale selektiert wurden.

Listing 4.6 zeigt eine äquivalente Implementierung zu obigen Beispiel unter Verwendung einer Entscheidungstabelle. Dieses hat den Vorteil, dass es sich auf einfache Weise um weitere Bedingungen und Regeln erweitern lässt. Beispielsweise kann so auch ein logisches nicht-exklusives Oder spezifiziert werden, wie in Listing 4.7 geschehen. Durch die beiden Regeln und der Verwendung des Bedingungsanzeigers „-“ wird Aktion 1 ausgeführt, sobald eine der beiden Bedingungen erfüllt ist. Auf diese Weise lassen sich auch komplexe Merkmalinteraktionen implementieren. Im Rahmen dieser Arbeit soll angenommen werden, dass hierarchische Abhängigkeiten zwischen Merkmalen durch geschachtelte `IF`-Abfragen repräsentiert werden und Constraints durch Entscheidungstabellen.

```
1 .DTSTART
2 .DT-A.EQ.1,      Y-
3 .DT-B.EQ.1,      -Y
4 .DT              XX,
5 .*Aktion 1
6 .DTEND
```

Listing 4.7: OR-Interaktion mittels Entscheidungstabelle

```
1 .DTSTART
2 .DT-01.EQ.1,     YY
3 .DT-02.EQ.1,     NY
4 .DT              XX,
5 .RC=8
6 .QUIT-MACRO
7 .DTEND
```

Listing 4.8: Constranits mittels Entscheidungstabelle

Als letztes soll gezeigt werden, wie Constraints im Quelltext abgebildet werden können. Wenn eine bestimmte Merkmalkombination vorliegt, darf kein Programm erzeugt werden. Um auf den Fehler hinzuweisen, bieten sich Rückgabewerte an. Diese werden mittels des Befehls `.RC=<n>` gesetzt, wobei n den Rückgabewert darstellt. Das ADS-Handbuch empfiehlt, den Wert 8 zu verwenden um deutlich zu machen, dass der generierte Quelltext Fehler enthält (vgl. [DSTG 2012a, 175]). Regel 1 der Entscheidungstabelle (Listing 4.8) implementiert eine requires-Abhängigkeit der Form `01 requires 02`. Wenn Merkmal `01`, aber nicht Merkmal `02` gewählt wurde, wird das Programm abgebrochen. Die Umsetzung einer excludes-Beziehung ist in Regel 2 gezeigt. Das Programm wird mit einer Fehlermeldung beendet, wenn beide Merkmale selektiert wurden.

5 Variabilitätsextraktion

5.1 Reverse Engineering

Der Begriff *Forward Engineering* bezeichnet den Prozess der Neuentwicklung eines Softwaresystems. Ausgangspunkt ist ein abstraktes Analysemodell, das durch iterative Konkretisierung zu einem Entwurfsmodell verfeinert wird, das dann die Grundlage der Implementierung darstellt (vgl. [Balzert 2011, 538]). In Kapitel 2 wurde beschrieben, wie die Abläufe bei der Entwicklung einer SPL aussehen.

Wird die Richtung des Entwicklungsprozesses umgekehrt, wird von *Reverse Engineering* gesprochen. Es umfasst zahlreiche Tätigkeiten, dazu gehören das Redokumentieren, Rekonstruieren, Respezifizieren sowie das Restrukturieren. Ihnen ist gemein, dass sie die Wartung und Weiterentwicklung erleichtern, in dem sie das Verständnis für das Altsystem erhöhen oder seine Qualität verbessern. Im Rahmen dieser Arbeit ist vor allem das Redokumentieren von Bedeutung: Durch eine Programmanalyse sollen Dokumente nachträglich erzeugt werden, die eigentlich beim Forward Engineering erstellt werden sollten, aber aus unterschiedlichen Gründen nicht zur Verfügung stehen. Zum Beispiel, weil nicht ausreichend dokumentiert wurde oder Modelle nicht aktualisiert wurden. Existiert beispielsweise nur ein veraltetes UML-Klassendiagramm, kann durch Redokumentieren versucht werden, die erforderlichen Informationen aus dem Quelltext zu extrahieren, um ein aktuelles Modell zu erzeugen. Im Zuge des Reverse Engineering kann jedoch auch der Quelltext überarbeitet werden. So werden beim Restrukturieren Transformationen durchgeführt, die Änderungen am Programm selbst vornehmen, aber stets semantische Äquivalenz gewährleisten. Zum Beispiel lässt sich auf diese Weise eine Software mit Sprunganweisungen so verändern, dass die `goto`-Anweisungen ersetzt werden ohne das Verhalten zu beeinflussen (vgl. [Balzert 2011, 538ff]).

Das Reverse Engineering kann auf verschiedene Weisen durchgeführt werden: So kann die Programmanalyse entweder statisch oder dynamisch erfolgen, je nachdem ob der Quelltext oder die laufende Anwendung untersucht wird. Zudem kann der Prozess entweder manuell, werkzeuggestützt oder sogar vollautomatisiert durchgeführt werden (vgl. [Balzert 2011, 538]).

5.2 Variabilität in Generatorsystemen

Bei der *Variabilitätsextraktion* handelt es sich um eine spezielle Form des Redokumentierens. Aus den Quellartefakten werden Variabilitätsinformationen extrahiert. Generatorsysteme bestehen aus vielen unterschiedlichen Artefakten, von denen ein Großteil über Variabilität verfügt. Im Folgenden soll näher untersucht werden, welchen Bestandteilen dabei welche Bedeutung zukommt.

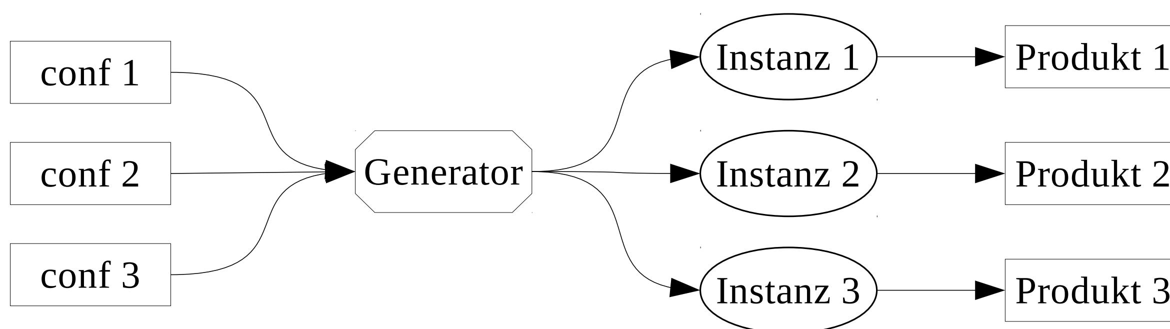

Abbildung 5.1: Zusammenhang zwischen statischen (rechteckig) und dynamischen Variabilitätsobjekten (oval) in einem Generatorsystem

Ein Programm kann statisch vorliegen, entweder kompiliert oder in Quelltextform. Sofern nur eine Version existiert, liegt keine räumlich-statische Variabilität vor. Durch das Ausführen des Programms entstehen dynamische Instanzen. Diese sind identisch, sofern die Umgebung und Aufrufparameter ebenfalls identisch sind. Durch das Verwenden unterschiedlicher Daten entstehen zur Laufzeit unterschiedliche Instanzen. Dieser allgemeine Sachverhalt gilt auch für Generatorsysteme, wie in Abbildung 5.1 dargestellt ist, wobei die Konfiguration den eben genannten Daten entsprechen.

Ein Generatorsystem setzt sich aus drei Hauptelementen zusammen: Der Konfiguration, dem Generator und den Produkten. Alle drei verfügen über Variabilität. Interessant sind aber auch die Beziehungen, die zwischen den Variationspunkten der Artefakte vorliegen. Die zeitliche Dimension der Variabilität soll dabei außer acht gelassen werden, da unabhängig vom Einsatz bestimmter Techniken nahezu jedes Dokument im Entwicklungsprozess einer Evolution unterliegt.

Da Konfigurationen in vielfacher Ausführung existieren, verfügt diese Menge über eine räumlich-statische Variabilität. Hingegen liegt der Generator nur in einer Ausführung vor. Wird er mit einer konkreten Konfiguration ausgeführt, transformiert er die darin enthaltenen Informationen, um die Produkte zu erstellen. Daher verfügen sowohl die Generatorinstanzen als auch die Produkte ebenfalls über räumliche Variabilität. Aus diesem Grund finden sich in der Literatur auch zahlreiche teils sehr unterschiedliche Ansätze. Sie unterscheiden sich neben dem Automatisierungsgrad in zwei Punkten: Zum einen verwenden sie verschiedene Artefakte als Informationsquelle, beispielsweise Generator-Konfigurationen oder Quelltexte von Produkten. Zum anderen extrahieren sie unterschiedliche Informationen. Lozano unterscheidet zwischen Merkmalen und ihren Abhängigkeiten, Varianten und Produkten (vgl. [Lozano 2011]). In den beiden folgenden Abschnitten soll ein kurzer Überblick über die verschiedenen Ansätze gegeben werden.

Bevor jedoch der Frage nachgegangen wird, wie Variabilitätsinformationen extrahiert werden können, muss zunächst überlegt werden, welche Informationen sich aus den Artefakten gewinnen lassen. Fast allen Ansätzen ist gemein, dass sie zunächst obligatorische und optionale Merkmale zu identifizieren versuchen. Auf dieser Basis können bereits einfache Merkmalmodelle erstellt werden. Diese würden jedoch nahezu jede Kombination von Merkmalen

erlauben, da noch keine Abhängigkeiten identifiziert wurden. Es müssen also weitere Informationen extrahiert werden. Aus diesem Grund verfügen einige Ansätze über die Fähigkeit, auch Untermerkmale und Constraints identifizieren zu können. Des Weiteren gibt es Methoden, die herausarbeiten, welche gültigen Konfigurationen es gibt und somit, welche Produkte sich generieren lassen und welche nicht (vgl. [Loesch/Ploedereder 2007, 1]). Die Unterscheidung zwischen den genannten Konzepten ist nicht sehr scharf, da es Überschneidungen gibt: Sind alle Merkmale und ihre Eigenschaften bekannt, kann daraus sowohl ein vollständiges Merkmalmodell erstellt werden, als auch eine Auflistung gültiger Konfigurationen in Tabellenform. Der Fokus ist jedoch ein anderer: Erstgenannter Ansatz konzentriert sich darauf, die Domäne zu verstehen, in dem die Variabilität modelliert wird. Die zweite Variante ist technischer ausgerichtet und stellt das Konfigurationswissen in den Vordergrund.

5.3 Bestehende Ansätze

Wie bereits erwähnt, stehen neben generativen Systemen eine Reihe weiterer Techniken zur Verfügung, um Variabilität zu implementieren. Ebenso vielfältig sind die Techniken, um die Variabilität aus den Artefakten zu extrahieren. Im Folgenden soll daher ein Überblick über einige bestehende Ansätze gegeben werden. Diese extrahieren die Variabilität aus verschiedenen Quellen und verfügen über einen unterschiedlichen Automatisierungsgrad (vgl. [She et al. 2012, 5]). Beispielsweise existieren manuelle Methoden, gegebenenfalls werkzeugunterstützt, um Variabilitätsmuster im Quelltext aufzuspüren (vgl. [Lozano 2011, 46]). Im Rahmen dieser Arbeit sind allerdings ausschließlich komplett automatisierbare Verfahren von Interesse.

Als Quelle der Variabilitätsinformationen eignen sich verschiedene Artefakte. Dazu zählen beispielsweise Konfigurationen, Abhängigkeiten zwischen Merkmalen sowie der Quelltext des Generators. Im Ansatz von Loesch und Ploedereder [2007] werden die bisher erstellten Produktkonfigurationen eines Generators ausgewertet. In den Konfigurationen steht aber nicht explizit, um welche Art Merkmal es sich handelt; diese Information muss erst durch eine Analyse verfügbar gemacht werden. Wenn zum Beispiel ein Merkmal in sämtlichen Ausführungen enthalten ist, wird daraus abgeleitet, dass es sich um ein obligatorisches Merkmal handelt. Kommt es stattdessen in nur einem Teil der Konfiguration vor, ist es entweder optional oder Bestandteil einer Merkmalgruppe. Es ist auch möglich, Constraints abzuleiten, beispielsweise lassen sich ausschließlich zusammen auftretende Merkmale auf requires-Beziehungen abbilden. Diese Schlussfolgerungen sind jedoch nur korrekt, wenn alle denkbaren Konfigurationen bekannt sind und untersucht werden können. Beispielsweise bildet die Menge an Konfigurationen, in der ein bestimmtes optionales Merkmal selektiert wurde, eine echte Teilmenge der Gesamtmenge an Konfigurationen.

Neben den Konfiguration können auch die Produkte zur Variabilitätsextraktion herangezogen werden. Die Grundidee ist ähnlich zum eben beschriebenen Ansatz: Die verschiedenen Ausführungen des Artefakts werden mit einander verglichen, um obligatorische und optionale Merkmale zu identifizieren. Antkiewicz et al. haben dies auf Anwendungssysteme an-

```
1 #define A
2 #ifdef A
3
4 // C++ -Code
5
6 #endif
```

Listing 5.1: *Version selection by #ifdef* mittels Präprozessoranweisungen

gewandt, die mit Hilfe eines Frameworks erstellt wurden. Dabei handelt es sich, wie bei den ADS-Makros, um Generatorsysteme. Es wird eine Merkmalhierarchie extrahiert die zeigt, welche Funktionalitäten des Frameworks auf welche Weise verwendet werden. Dabei muss nicht der gesamte Quellcode analysiert werden, sondern lediglich die Stellen, an denen Anwendung und Framework miteinander interagieren. Dies ist zum Beispiel beim Aufruf von Funktionen oder Datentypen des Frameworks der Fall (vgl. [Antkiewicz et al. 2009, 2ff]). Diese Abschnitte stellen somit Variationspunkte dar. Anschließend werden die so identifizierten Codeabschnitte zwischen den einzelnen Anwendungen auf Gemeinsamkeiten und Unterschiede untersucht, um die Beziehungen zwischen den einzelnen Merkmalen zu bestimmen.

Werden die Variabilitätsinformationen aus dem Quelltext extrahiert, sind die Ansätze daher oft auf eine spezielle Implementierungsweise zugeschnitten. Beispielsweise beschreiben Parra et al. ein Verfahren, dass aspektorientierte Programmierung voraussetzt (vgl. [Parra et al. 2010, 1ff]). Solche Methoden sind jedoch für diese Arbeit nicht relevant und sollen daher nicht näher behandelt werden. Ein weiterer Ansatz beschäftigt sich mit der Variabilitätsextraktion aus Generatorsystemen, die mithilfe des C++-Präprozessors realisiert sind (vgl. [Snelting 1996, 146]). Dieser erlaubt es, vor dem Kompilieren des Quelltexts Makros und Präprozessoranweisungen auszuführen. Der Präprozessor wird unter anderem genutzt, um Dateien zu inkludieren, kann aber auch zur bedingten Kompilierung eingesetzt werden. Dabei werden bestimmte Abschnitte des Quelltexts von der Kompilierung ausgeschlossen. Gesteuert wird dies durch das Definieren von Makros, wie in Listing 5.1 gezeigt.

Der C++-Code, der sich zwischen dem `#ifdef` und `#endif` befindet, wird nur kompiliert, wenn zuvor das Makro `A` definiert wurde. Dies ist in Zeile 1 mittels `#define` geschehen. Die definierten Makros stellen somit die Konfiguration dar und entscheiden, über welche Bestandteile das Produkt verfügt. Es wird nun davon ausgegangen, dass jedes dieser Makros ein Merkmal repräsentiert. Demzufolge entspricht das Makro `A` in Listing 5.1 einem optionalen Merkmal. Durch Analyse der verwendeten Makros sowie ihren Beziehungen, die sich durch logische Operatoren ergeben, werden Tabellen erstellt, die Auskunft über gültige Konfigurationen geben und den Ausgangspunkt weiterer Untersuchungen bilden (vgl. [Snelting 1996, 148ff]). Beispielsweise wird versucht die Konfiguration zu vereinfachen und ungewollte Abhängigkeiten zwischen Merkmalen zu entdecken (vgl. [Snelting 1996, 153]). Der Ansatz unterliegt jedoch einigen Limitierungen. Zum einen wird gefordert, dass ausschließlich die drei in Listing 5.1 verwendeten Präprozessoranweisungen verwendet werden.

Dies wird als Paradigma „*version selection by #ifdef*" [Snelting 1996, 147] bezeichnet. Zum anderen wird davon ausgegangen, dass jede `#ifdef`-Anweisung zur Implementierung von Variabilität dient. Dies ist jedoch nicht gegeben. Beispielsweise kann es der Fall sein, dass in Abhängigkeit des verwendeten Compilers unterschiedliche Bibliotheken inkludiert werden müssen. Hierzu kann ebenfalls `#ifdef` verwendet werden. Es handelt sich jedoch nicht um ein Merkmal, sondern um ein Implementierungsdetail.

Darüber hinaus gibt es noch Ansätze, die das Laufzeitverhalten untersuchen und zum Beispiel durch den Zugriff auf Daten versuchen, Merkmale und Untermerkmale zu identifizieren. Eine nähere Beschreibung dieser Verfahren findet sich bei Yang et al. [2009] sowie Egyed [2001] . Diese Ansätze sind für die vorliegende Anwendung ungeeignet, da mangels Konfiguration auch keine Produkte erzeugt werden können.

5.4 Übertragung und Erweiterung bestehender Ansätze auf Generatorsysteme

Bei der Betrachtung der in Abschnitt 5.3 vorgestellten Ansätze zur Variabilitätsextraktion wird ersichtlich, dass keiner direkt auf das Szenario dieser Arbeit anwendbar ist. Einige Verfahren untersuchen Generatorsysteme, jedoch nicht den Quelltext des Generators sondern beispielsweise die Konfigurationen. Andere Ansätze führen zwar eine statische Analyse des Generatorcodes durch, basieren aber auf anderen Variabilitätsimplementierungen. Lediglich das Anwendungsszenario bei Snelting ist mit ADS-basierten Generatoren vergleichbar. Es wird Quelltext analysiert, der später Anwendungscode erzeugt. Es existieren jedoch auch Unterschiede: In ADS werden Schleifen, Berechnungen, Methodenaufrufe und Variablen verwendet, sodass das Paradigma „*version selection by* `#ifdef`" nicht zutrifft. Daher gestaltet sich die Merkmalerkennung deutlich aufwendiger als im Falle des Präprozessors. Würde man Sneltings Ansatz direkt auf ADS-Makros übertragen, würde dies folgendes bedeuten: Aufgrund der Einschränkung auf wenige Sprachkonstrukte könnte nur ein Bruchteil der Variabilität extrahiert werden. Des Weiteren gibt es keine Techniken, um delokalisierte Merkmale oder Merkmalinteraktionen zu identifizieren. Das Ergebnis wäre daher äußerst unbefriedigend. Aus diesem Grund sollen einige theoretische Vorüberlegungen angestellt werden, um den Ansatz auch auf ADS-Makros anwenden zu können.

Es ist nicht zielführend, jedes Sprachkonstrukt von ADS separat zu behandeln, da zu viele Fälle betrachtet werden müssten. Aus diesem Grund soll von den konkreten Anweisungen abstrahiert und stattdessen auf sogenannten Kontroll- und Datenflussgraphen gearbeitet werden. Diese enthalten Informationen über die Reihenfolge, in der die einzelnen Befehle ausgeführt werden, sowie über Lese- und Schreibzugriffe auf Variablen. Sie werden in den Abschnitten 5.5.2 und 5.5.3 näher vorgestellt. Die Arbeit auf Graphen bringt mehrere Vorteile mit sich: Zum einen verringert sich so die Anzahl der zu untersuchenden Möglichkeiten. Eine Verzweigung des Kontrollflusses kann gleichermaßen durch eine `IF`- oder `SELECT`-Anweisung hervorgerufen werden, ebenso wie durch eine Entscheidungstabelle. Für die Variabilitätsextraktion ist es jedoch unerheblich, welche Kontrollflussstruktur dem zugrunde liegt. Es ist somit nicht notwendig, alle drei Arten einzeln zu behandeln. Stattdessen genügt

es, im Datenflussgraphen bestimmte Muster zu erkennen und diese in ein Merkmalmodell zu transformieren. Zum anderen wird so eine Sprachabhängigkeit der Variabilitätsextraktion verringert. Da die Transformationen auf dem Graphen stattfinden, spielen konkrete Sprachkonstrukte keine Rolle mehr. Ob der Datenflussgraph ursprünglich beispielsweise aus Java- oder ADS-Code erzeugt wurde, ist nicht von Bedeutung. Sofern beide Sprachen über die selbe abstrakte Syntax verfügen, wird durch die Verwendung der ASTs als Eingabe dasselbe Abstraktionsniveau erreicht.

Des Weiteren ist es notwendig, delokalisierte Merkmale zu erkennen und die zugehörigen Codefragmente zu identifizieren. Hierfür kann ebenfalls auf dem Datenflussgraphen gearbeitet werden, wie in Abschnitt 5.8 gezeigt wird. Ein weiteres Ziel besteht darin, zu entscheiden, ob eine Anweisung der Implementierung von Variabilität dient oder nicht. Hierfür konnte in der Literatur kein geeigneter Ansatz gefunden werden. Aus diesem Grund wurde vom Verfasser eine regelbasierte Unterteilung erarbeitet, um diese Limitierung bestehender Ansätze aufzuheben. In Abschnitt 5.5.4 wird hierauf näher eingegangen.

5.5 Statische Code-Analyse

Im vorangegangenen Abschnitt wurde erläutert, welche Variabilitätsinformationen ein Software-Generator enthält. Nun sollen die Grundlagen einiger Techniken vorgestellt werden, mit denen diese Informationen extrahiert werden können. Dazu wird im folgenden Abschnitt zunächst der Kontroll- und Datenflussgraph eingeführt. Dabei handelt es sich um Hilfsartefakte, auf denen die Kontroll- und Datenflussanalysen ausgeführt werden. Diese werden anschließend in den Abschnitten 5.5.2 und 5.5.3 behandelt.

5.5.1 Abstrakter Syntaxbaum

Die zu untersuchenden Makros liegen bereits in einer verarbeiteten Version vor, nämlich in Form eines AST. Dabei handelt es sich um eine Baumdarstellung der abstrakten Syntax eines Programms. Wird im Allgemeinen von der Syntax gesprochen, ist damit in den meisten Fällen die sogenannte *konkrete Syntax* gemeint. Jede Programmiersprache verfügt über eine für sie spezifische Syntax. Sie gibt an, wie ein formal korrektes Programm auszusehen hat. Parser verwenden die konkrete Syntax um Strukturen im Programm zu erkennen und darauf reagieren zu können. Die abstrakte Syntax verzichtet auf Informationen, die zur Beschreibung der Programmstruktur nicht benötigt werden. Dazu zählen unter anderem Klammern und Variablennamen.

Zur Verdeutlichung soll der ADS-Ausdruck `IF.01.EQ.A` betrachtet werden. Dieser lässt sich in einen AST transformieren, der ausschnittsweise in Listing 5.2 gezeigt ist. Der selbe Ausdruck wird in Form von Extensible Markup Language (XML)-Elementen wiedergegeben.

Ein AST ist somit lediglich die Repräsentation des Quelltexts in einer allgemeineren Darstellungsform. Er kann zum Beispiel durch die Sprache *Turing Extender Language (TXL)*

```
<if_statement>
    IF-
    <parameter_new>
        <LocalParameter>01</LocalParameter>
    </parameter_new>
    .
    <op>EQ</op>
    .
    <ads_expression>
        <repeat_no_comma>
            <no_comma>
                <no_newline>
                    <id>A</id>
                </no_newline>
            </no_comma>
            <empty />
        </repeat_no_comma>
    </ads_expression>
</if_statement>
```

Listing 5.2: Ausschnitt eines AST einer IF-Anweisung

erzeugt werden. Die Sprache wurde speziell für Sprachtransformationen konzipiert. Mit ihrer Hilfe lassen sich Parser realisieren, die automatisch den Eingabequelltext in einen AST transformieren können. Damit dies möglich ist, muss jedoch folgendes bekannt sein: Zum einen die Grammatik und die Syntax der Ausgangssprache. Die Transformation ist somit für jede Programmiersprache spezifisch; für jede muss die Grammatik separat beschrieben werden. Zum anderen ist es notwendig, die abstrakte Syntax zu spezifizieren. Auch wenn diese allgemein und unabhängig von der konkreten Syntax ist, so gibt es dennoch nicht *die* abstrakte Syntax, sondern die konkrete Implementierung ist weiterhin beliebig. Im vorliegenden Fall erzeugt der TXL-Parser eine XML-Datei, wie in Listing 5.2 ausschnittsweise gezeigt. Dabei muss jede mögliche Variabilität in der konkreten Syntax berücksichtigt werden. Aus diesem Grund besteht beispielsweise der XML-basierte Syntaxbaum eines lediglich zehnzeiligen Makros aus mehreren hundert XML-Elementen.

Die Vorteile eines AST sind vielfältig: Die abstrakten Syntaxen verschiedener Programmiersprachen können einfacher vereinheitlicht werden, als dies bei reinem Quelltext der Fall ist. Dies ermöglicht den Vergleich verschiedener Programmiersprachen (vgl. [Würsch 2006, 13f]). Beispielsweise können statische Code-Metriken auf Basis des AST erhoben werden und nicht nur für eine bestimmte Sprache. Weiterhin werden ASTs verwendet, um Duplikate im Code aufzuspüren oder um verschiedene Versionen einer Software miteinander zu vergleichen. Zudem finden sie in Anwendung in der Sprachtransformation, also wenn ein Quelltext in eine andere Programmiersprache überführt werden soll. Dazu wird der Quelltext der Ausgangssprache zuerst in einen AST umgewandelt, bevor dieser in die Zielsprache transformiert wird.

5.5.2 Kontrollflussanalyse

Der Kontrollfluss einer Software zeigt an, welche Anweisungen in welcher Reihenfolge ausgeführt werden können (vgl. [Balzert 1998, 399]). Zur Darstellung werden häufig Kontrollflussgraphen verwendet, wie in Abbildung 5.2 dargestellt. Zum besseren Verständnis wurde daneben der Ausgangsquelltext, eine einfache Java-Funktion, platziert. Bei Kontrollflussgraphen handelt sich um gerichtete Graphen. Jede Anweisung im Quelltext wird durch einen eigenen Knoten repräsentiert. Die ausgehenden Kanten eines Knotens geben an, welche Knoten als nächstes ausgeführt werden können. Beispielsweise entspricht Knoten n_2 aus Abbildung 5.2 einer `IF`-Abfrage. Danach wird entweder die Anweisung n_3 oder n_4 ausgeführt, je nachdem ob der `THEN`- oder der `ELSE`-Zweig zur Ausführung kommt. Durch die Verwendung von Schleifen können auch Zyklen entstehen, wie er zum Beispiel von den Knoten $n5$ und n_6 gebildet wird. Abgeschlossen wird ein Kontrollfluss durch einen Knoten n_{final}. Dieser besitzt keinen Nachfolger. Ein Kontrollflussgraph kann über mehrere derartige Knoten verfügen, so wie auch ein Programm über mehrere `return`-Anweisungen verfügen kann. Jede abwechselnde Folge von Knoten und Kanten, die mit n_1 beginnt und mit n_{final} endet, heißt Pfad (vgl. [Balzert 1998, 399]).

Besonders erläutert werden soll noch die Darstellung von Entscheidungstabellen in einem Kontrollflussgraph, da viele andere Sprachen nicht über ein vergleichbares Konstrukt verfügen. Jede Aktion soll aus Gründen der Übersichtlichkeit als ein Knoten dargestellt werden. Dies stellt eine Aggregation dar, da eine Aktion mehrere Anweisungen umfassen kann. Die Aktionen werden in der Reihenfolge ausgeführt, in der sie definiert wurden. Aktion A_1 wird daher immer vor A_2 ausgeführt, gefolgt von $A3$ und so weiter. Dies entspricht der Darstellung in Abbildung 5.3. Vor jeder Aktion wird anhand der Bedingungsanzeiger B_n geprüft, ob sie ausgeführt wird. Ansonsten wird die nächste Aktion überprüft. Dies ermöglicht eine übersichtliche Darstellung, entspricht allerdings nicht dem Verhalten von ADS. Dort erfolgt nur eine Auswertung der Bedingungen und anschließend werden die passenden Aktionen ausgeführt. Dies führt jedoch zu einem schwer verständlichen Graphen, da für mögliche Kombination von Aktionen ein Pfad vorhanden sein muss. Schon bei nur drei Aktionen existieren so viele Kanten, dass Überlappungen nicht vermieden werden können. Es wurde sich daher für die in Abbildung 5.3 gezeigte Variante entschieden. Auch wenn sie nicht identisch ist mit der ADS-internen Verarbeitung von Entscheidungstabellen, so führt sie doch zu den gleichen Resultaten.

Wird ein externes Makro aufgerufen, gibt es zwei Möglichkeiten: Zum einen kann einfach ein Knoten angelegt werden, der den Makroaufruf repräsentiert. Zum anderen kann der Kontrollflussgraph des aufgerufen Makros an die entsprechende Stelle eingefügt werden. Dies setzt voraus, dass auch das aufgerufene Makro in Form eines AST vorliegt. Der Vorteil liegt darin, dass auf diese Weise auch die aufgerufenen Makros analysieren werden können. Sofern möglich, soll daher stets die zweite Variante angewandt werden.

Der Kontrollfluss gibt Auskunft über die Struktur einer Software und kann direkt aus dem AST abgeleitet werden. Die Kontrollflussanalyse ist eine Methode der statischen Codeanaly-

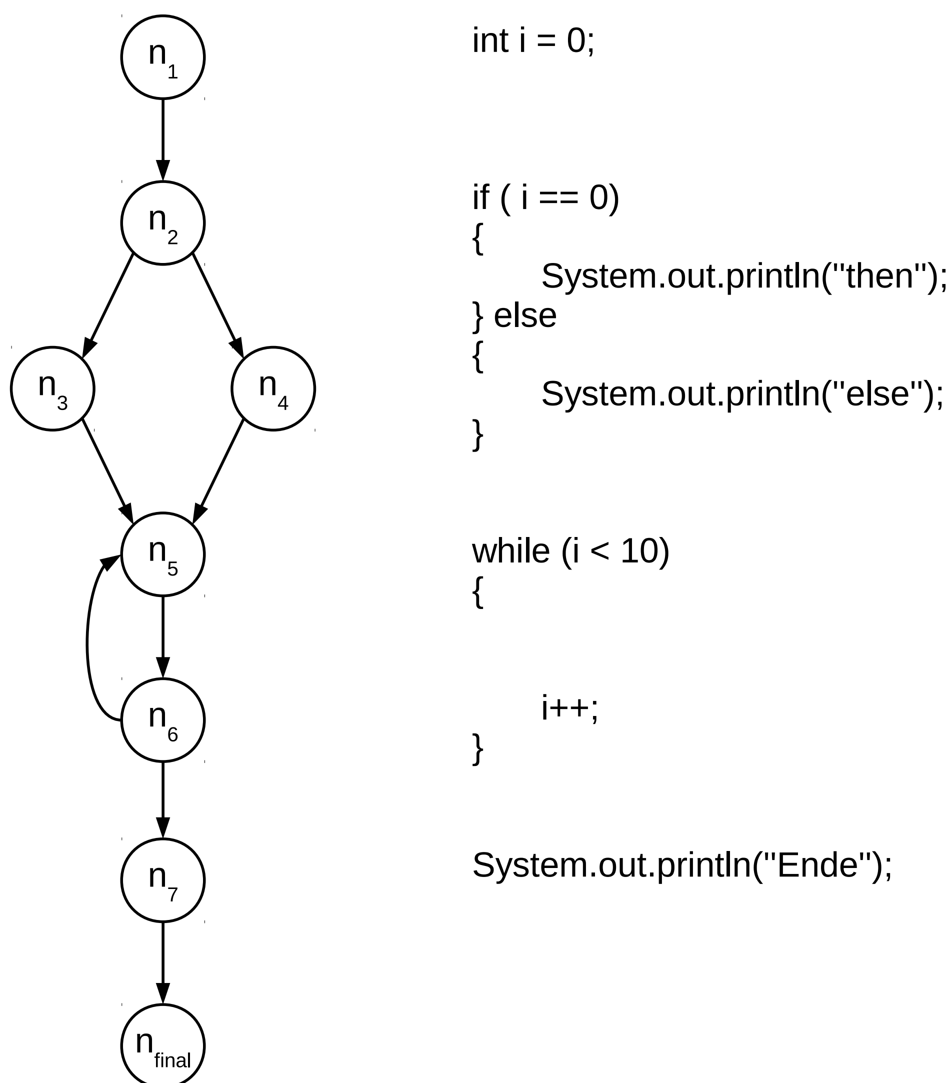

Abbildung 5.2: Kontrollflussgraph

se und trifft daher keine konkreten Aussagen über das Laufzeitverhalten. Es kann also nicht gesagt werden, welcher Knoten nun tatsächlich nach n_2 ausgeführt wird oder nach wie vielen Ausführungen die Schleife abgebrochen wird. Durch den Kontrollflussgraph werden nur die prinzipiellen Möglichkeiten beschrieben. Es kann auch formuliert werden, dass der Kontrollflussgraph die Summe der Kontrollflüsse aller möglichen Instanzen darstellt.

Kontrollflussgraphen werden unter anderem zur Berechnung statischer Code-Metriken verwendet. Dabei handelt es sich um Kennzahlen, die auf Basis des Quelltexts berechnet werden. Beispielsweise gibt die Zweigüberdeckung eines Programms an, welche Abschnitte des Systems durch Softwaretests überprüft werden (vgl. [Balzert 1998, 400]). Bei der Verwendung von Kontrollflussgraphen sind in vielen Fällen die Verzweigungen und Pfade von Interesse. Daher werden aufeinander folgende Knoten, die keine Verzweigung aufweisen, häufig zusammengefasst um eine kompaktere Darstellung zu erreichen. Dies hat keinen Einfluss auf die Anzahl der möglichen Pfade oder Verzweigungen. Ein ähnlicher Ansatz soll in Abschnitt 5.6 genutzt werden, um Merkmale zu identifizieren. Dabei werden Knoten, die dasselbe Merkmal implementieren, zu einer Sequenz zusammengefasst. Darüber hinaus werden

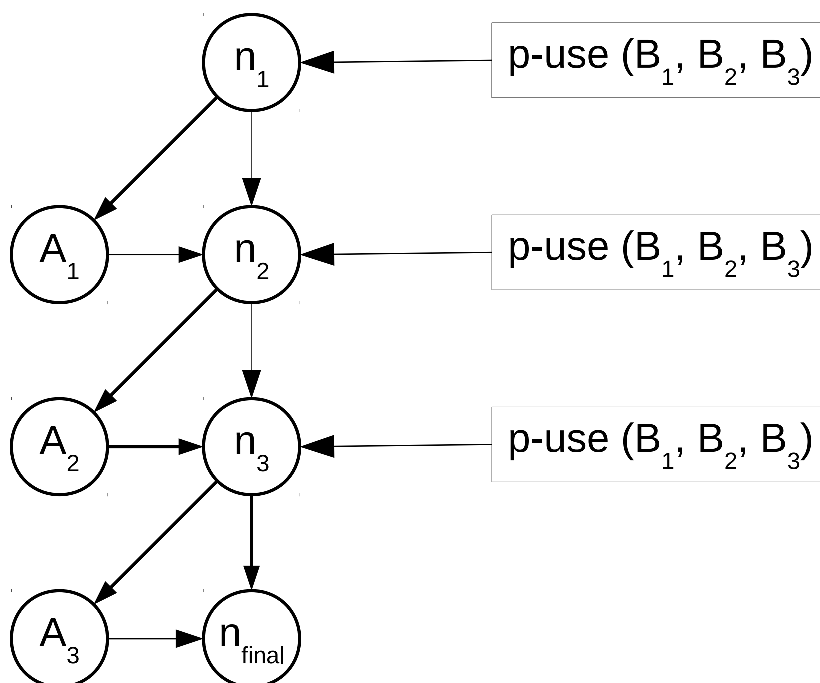

Abbildung 5.3: Kontrollflussgraph einer Entscheidungstabelle

Kontrollflussgraphen in den Abschnitten 5.7 und 5.8 eingesetzt, um Beziehungen zwischen Merkmalen zu erkennen.

5.5.3 Datenflussanalyse

Im vorangegangenen Abschnitt wurde die Kontrollflussanalyse vorgestellt. Der Kontrollfluss allein ist jedoch nicht aussagekräftig genug. Da keine Variablen betrachtet werden, ist es auch nicht möglich, Zusammenhänge zwischen ihnen zu entdecken. Daher können auch keine Zusammenhänge zwischen entfernten Code-Abschnitten gewonnen werden. Die Datenflussanalyse stellt eine Erweiterung der Kontrollflussanalyse dar. Der Datenfluss eines Programms gibt an, zu welchem Zeitpunkt Daten, das heißt in der Regel Variablen, gelesen und und geschrieben werden. Die Darstellung erfolgt in Form eines Datenflussgraphen (vgl. [Balzert 1998, 421]). Dieser ähnelt dem Kontrollflussgraphen, enthält aber zusätzlich Informationen zu Lese- und Schreibzugriffen. Abbildung 5.4 zeigt den Datenflussgraphen des im vorangegangenen Abschnitt verwendeten Beispiels.

Jedes Mal, wenn einer Variable ein Wert zugewiesen wird, und sei es der bestehende, wird dies als schreibender Zugriff gewertet. Im Graphen werden diese durch das Schlüsselwort *def* gekennzeichnet, gefolgt vom Variablennamen. In vielen Programmiersprachen gibt es mehrere Möglichkeiten der Wertzuweisung. Bei einem Datenflussgraphen wird jedoch nicht zwischen ihnen differenziert, da es die Struktur nicht beeinflusst. Der Lesezugriff unterteilt sich hingegen in zwei Unterarten. Diese werden als *p-use* und *c-use* bezeichnet (vgl. [Balzert 1998, 421]). Maßgeblich für die Unterscheidung ist der Zweck des Zugriffs. Wird der Wert der Variablen dazu verwendet einen anderen Wert zu berechnen (der gegebenenfalls wieder einer Variable zugewiesen wird) handelt es sich um einen c-use. In Abbildung 5.4 tritt dies

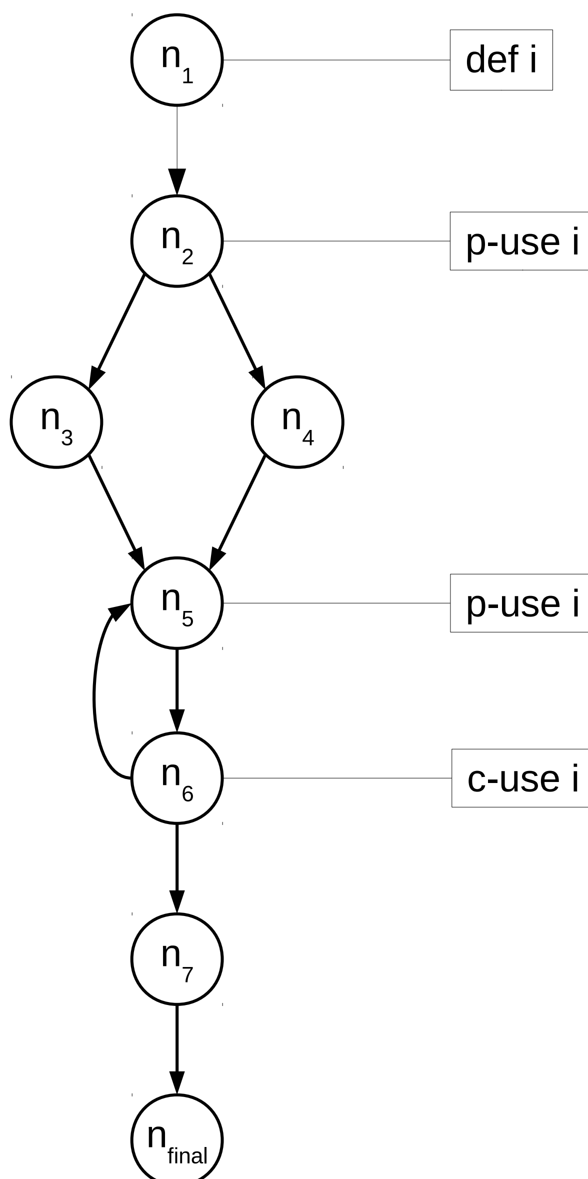

Abbildung 5.4: Datenflussgraph

bei Knoten n_6 auf, wenn der Wert aus Variable `i` gelesen wird. Hat der Wert Einfluss auf den Kontrollfluss handelt es sich um einen Zugriff des Typs p-use. Dies ist zum Beispiel bei der Überprüfung einer `if`-Bedingung wie bei Knoten n_2 der Fall. Im Laufe ihres Lebenszyklus' kann eine Variable beliebig gelesen und überschrieben werden. Dabei sind beliebige Kombinationen der drei Zugriffsarten möglich, mit einer Ausnahme: Bevor eine Variable gelesen wird, sollte mindestens ein Schreibzugriff erfolgt sein, da andernfalls unerwünschtes Verhalten auftreten kann. Hierzu ist es allerdings notwendig, den Variablennamen zu kennen, um zu wissen, auf welche Variable ein Zugriff erfolgt ist. Daher kann eine Datenflussanalyse nicht ausschließlich auf einen abstrakten Syntax arbeiten, sondern benötigt ebenfalls den ursprünglichen Quelltext. Die ASTs der ADS-Makros enthalten beides, sodass sich hieraus kein Problem ergibt.

Datenflussgraphen haben ein ähnliches Anwendungsgebiet wie Kontrollflussgraphen. Sie ermöglichen es, Anweisungen zu erkennen, die sich an verschiedenen Stellen des Quelltexts

befinden, aber von denselben Variablen abhängen. Auf diese Weise kann eine Datenflussanalyse zur Erkennung der Fragmente delokalisierter Merkmale sowie weiterer Merkmalinteraktionen herangezogen werden. Diese Verfahren werden in Abschnitt 5.8 beschrieben.

5.5.4 Bedeutung von Variablen

Sollen im Quelltext Merkmale identifiziert werden, muss bekannt sein, wie diese im Code repräsentiert sind. Zwar können dazu Variablen eingesetzt werden, doch hat nicht jede Variable eines Generators dieselbe Funktion. Es wird daher eine Unterscheidung nach drei Kategorien vorgeschlagen, die auf den verschiedenen Einsatzmöglichkeiten von Variablen aufbauen. Zum einen können Variablen direkt ein Merkmal beziehungsweise eine Merkmalgruppe repräsentieren. Diese Art soll im weiteren Verlauf als *Merkmalvariable* bezeichnet werden. Zum anderen existieren *Hilfsvariablen*, die keinem Merkmal zugeordnet sind, sondern beispielsweise zur Speicherung von Zwischenergebnissen dienen. Weiterhin gibt es Variablen, die zur Überprüfung und Steuerung von Merkmalinteraktionen dienen. Diese sollen im Folgenden als *Interaktionsvariablen* bezeichnet werden. Da es sich bei dieser Kategorisierung um keine etablierte, sondern um eine vom Verfasser gewählte Unterteilung handelt, sollen in den folgenden Abschnitten eine ausführlichere Erklärung der drei Variablenarten erfolgen.

Merkmalvariablen

Variablen können direkt einem Merkmal entsprechen. In diesem Fall kann beispielsweise eine Variable existieren, die auf 1 gesetzt wird, wenn das Merkmal selektiert wurde und die andernfalls den Wert 0 annimmt. Zudem ist es möglich die Merkmalgruppe, zum Beispiel *Mobilfunkstandard*, in einer Variable abzubilden. Anschließend kann sie mit passenden Werten belegt werden, die angeben, welche Merkmale selektiert wurden. Je nach Programmiersprache stehen gegebenenfalls weitere Implementierungsmöglichkeiten wie Bitmasken zur Verfügung. In jedem Fall enthalten die entsprechenden Variablenwerte Variabilitätsinformationen.

Die Bedeutung der Variablen soll anhand des Beispiels des Frequenzwechslers gezeigt werden, dass bereits in Abschnitt 3.3.4 vorgestellt wurde. Aus Gründen der Übersichtlichkeit wird sich jedoch auf die beiden Mobilfunkstandards LTE und UMTS beschränkt. Dieses Beispiel wird in den nächsten beiden Abschnitten erweitert. Zunächst wird gezeigt, wie die Merkmale im Quelltext repräsentiert werden können. Dies ist in Listing 5.3 geschehen.

Die Variable `O1` steht für die Auswahl des UMTS-Merkmals und `O2` für LTE. Sollen Anweisungen nur im Falle der Selektierung eines bestimmten Merkmals ausgeführt werden, so kann der Variableninhalt überprüft werden, wie hier mit einer `IF`-Abfrage geschehen. Je nach Ergebnis der Evaluierung können dann unterschiedliche Anweisungen ausgeführt werden. Eine Merkmalvariable V_M ist daher jede Variable die anzeigt, ob ein oder mehrere Merkmale ausgewählt wurden, oder nicht.

Die Wertzuweisung muss somit unmittelbar von äußeren Einflussfaktoren abhängen. Dies kann auf verschiedene Weise geschehen: Zum einen können dem Makro Aufrufparameter

```
1 .IF.01.EQ.1
2 .* LTE-spezifische Anweisungen
3 .IFEND
4
5 .IF.02.EQ.1
6 .* UMTS-spezifische Anweisungen
7 .IFEND
```

Listing 5.3: Merkmalvariablen

übergeben werden. Diese enthalten die Informationen darüber, ob ein Merkmal selektiert wurde oder nicht. Zum anderen können Variablen gesetzt werden, in dem eine entsprechende Konfigurationsdatei eingelesen wird. Ausschließlich Variablen, bei denen die Wertzuweisung auf eine der beiden beschriebenen Weisen erfolgt, werden als Merkmalvariablen behandelt.

Allerdings gibt es Situationen, in denen eine Merkmalvariable auf anderem Wege gesetzt wird. Dies ist der Fall, wenn Vorgabewerte verwendet werden. Gibt der Benutzer beispielsweise keine Architektur an, wird automatisch eine 32-Bit-Architektur gewählt. Dadurch wird die Variable als Hilfsvariable erkannt und nicht als Merkmalvariable, um die es sich eigentlich handelt. Es konnte keine Möglichkeit gefunden werden, diesen Fall durch zusätzliche Regeln abzudecken.

Interaktionsvariablen

Im Abschnitt 2.5 wurde auf das Vorhandensein von Wechselwirkungen eingegangen, die bei bestimmten Merkmalkombinationen auftreten. Sofern ein Merkmal durch eine Variable repräsentiert wird, wie im vorangegangenen Abschnitt beschrieben, so müssen Merkmalkombinationen eine Verknüpfung von Merkmalvariablen darstellen. Diese sollen als Interaktionsvariablen bezeichnet werden, auch wenn diese Verknüpfung nicht explizit in einer Variable gespeichert werden muss. In modernen Programmiersprachen werden logische Operatoren eingesetzt, um Variablen zu verbinden, beispielsweise durch `C = A && B`. Da ADS allerdings solche Operatoren nicht kennt, bieten sich stattdessen Entscheidungstabellen an. Das Wesentliche ist jedoch die logische Verknüpfung der beiden Merkmalvariablen. Die Entscheidungstabelle stellt nur eine Möglichkeit dar, diese auszudrücken. Daher soll der Begriff Interaktions*variable* gebraucht werden, obwohl eine eigenständige Variable, welche die Verknüpfung repräsentiert, nicht existiert. Stattdessen soll jede Regel einer Entscheidungstabelle als anonyme Variable aufgefasst werden.

Im Folgenden soll das Beispiel der Merkmalvariablen erweitert werden: Sofern ein Smartphone mehrere Mobilfunkstandards parallel unterstützt, muss es über einen zusätzlichen Frequenzwechsler verfügen. Die bisherige Implementierung berücksichtigt noch nicht die Merkmalinteraktion. Durch die Verknüpfung der beiden Merkmalvariablen können interaktionsspezifische Anweisungen ausgeführt werden, wie in Listing 5.4 geschehen. Eine Interaktionsvariable V_I ist eine Variable, welche die Beziehungen zwischen zwei oder mehr Merkmalen ausdrückt. Sie wird zur Steuerung des Kontrollflusses benutzt, um Anweisungen nur

```
1 .DTSTART
2 .DT-01.EQ.1,     Y
3 .DT-02.EQ.1,     Y
4 .DT              X,
5 .* Anweisungen, wenn UMTS und LTS ausgewählt wurde
6 .DTEND
```

Listing 5.4: Implementierung einer Merkmalinteraktion mithilfe einer Entscheidungstabelle

```
1 .IF.01.EQ.1
2 .SET-03=1920
3 .IFEND
4 .* ...
5 .IF.03.GT.2000
6 .* Anweisungen in Abhängigkeit der Frequenz
7 .IFEND
```

Listing 5.5: Hilfsvariablen

im Falle bestimmter Merkmalkombinationen auszuführen. Interaktionsvariablen sind Variablen, welche die Beziehungen zwischen zwei oder mehr Merkmalen ausdrücken. Sie werden zur Steuerung des Kontrollflusses benutzt, um Anweisungen nur im Falle bestimmter Merkmalkombinationen auszuführen. Sie setzen sich stets aus mehreren Merkmalvariablen zusammen. Ob die Verknüpfung durch Entscheidungstabellen oder logische beziehungsweise arithmetische Operatoren erfolgt, ist unerheblich. Jede Variable, die sich aus Merkmalvariablen zusammensetzt, soll daher als Interaktionsvariable eingestuft werden.

Hilfsvariablen

Neben Merkmal- und Interaktionsvariablen existieren auch Variablen, die keine direkte Entsprechung in Form eines Merkmals haben. Sie sollen im Folgenden als Hilfsvariablen V_H bezeichnet werden. Darunter fallen beispielsweise Zählvariablen einer Schleife und Variablen, in denen die Ergebnisse von Berechnungen zwischengespeichert werden. Also all jene Variablen, die auch in einem herkömmlichen Einzelsystem zum Einsatz kommen. Ihre Belegung wird nicht direkt durch die Konfiguration des Produkts gesteuert. Stattdessen sind sie ein notwendiger Bestandteil der Verarbeitungsanweisungen. Entsprechend der in Abschnitt 3.2 gegebenen Definition der dynamischen Variabilität sind auch Hilfsvariablen Träger von Variabilitätsinformationen. Dies ist aber nur die Folge davon, dass bereits an anderer Stelle im Quelltext Variabilität aufgetreten ist, wie im folgenden Code-Beispiel gezeigt werden soll:

Je nach gewähltem Mobilfunkstandard wird eine andere Frequenz benutzt. Diese wird in Listing 5.5 in der Variablen 03 gespeichert. Sie hat also einen anderen Wert, wenn ein anderer Standard gewählt wurde. Beim Vergleich der Datenflüsse der beiden Instanzen ist daher ein Unterschied an diesem Knoten feststellbar. Dennoch stellt die Variable `03`, im Gegensatz zu `01` kein eigenständiges Merkmal dar. Beide Variablen enthalten dieselbe Variabilitätsinfor-

mation. In einem Merkmalmodell braucht diese nur einmal modelliert zu werden, da sie zur Implementierung eines anderen Merkmals gehört und es sich somit um redundante Informationen handelt.

Hilfsvariablen werden nicht direkt durch Aufrufparameter oder Konfigurationsdateien gesetzt. Es gibt eine Fülle von Möglichkeiten, wie die Wertzuweisung erfolgen kann. Beispielsweise können sie mit einem konstanten Wert initialisiert werden, Zählvariablen wird häufig der Wert 0 zugewiesen oder sie enthalten das Ergebnis der Berechnung. Es würden jedoch zu viele Regeln aufgestellt werden müssen, um alle Fälle erfassen zu können. Deshalb soll jede Variable als Hilfsvariable behandelt werden, die weder als Merkmal- noch als Interaktionsvariable erkannt wurde.

5.6 Merkmalerkennung

Um die im Quelltext vorhandenen Variabilitätsinformationen in einem Merkmalmodell darstellen zu können, müssen zuerst alle Merkmale identifiziert werden. Dabei ist nicht nur interessant, welche Merkmale vorhanden sind, sondern auch, in welchen Zeilen des Quelltexts sie implementiert sind. Zu diesem Zweck soll eine vollständige Traversierung des Datenflussgraphen erfolgen. Hierfür bietet sich die Tiefensuche (engl. *Depth-first Search, DFS*) an. Dabei handelt es sich um einen Knotenbesuchsalgorithmus, der sich im besonderen Maße eignet, um Informationen über die Struktur des Graphen zu erfahren (vgl. [Ottmann/Widmayer 1996, 554]). Die Nachfolger eines Knotens werden hierbei stets vor seinen noch unbesuchten Geschwisterknoten besucht. Wird dieser auf den Graphen aus Abbildung 5.2 angewandt, ergibt sich folgende Reihenfolge: $[n_1, n_2, n_3, n_5, n_7, n_4]$ Sobald ein neuer Knoten n gefunden wurde, soll er einer Sequenz S_m zugewiesen werden. Dabei handelt es sich um Teilgraphen, die aus einer Menge zusammenhängender Knoten und zugehöriger Kanten besteht. Anschließend soll eine Zuordnung zwischen Sequenzen des Graphen und Implementierungskomponenten erfolgen. Das Ziel besteht darin, dass alle Anweisungen, die dasselbe Merkmal implementieren, auch der selben Sequenz zugewiesen werden. Dennoch liegt keine $1:1$-Beziehung zwischen Sequenz und Implementierungskomponenten vor, wie noch erläutert wird.

Knoten die stets nacheinander ausgeführt werden, sind ein und derselben Sequenz zuzuordnen. Das ist bei einer Sequenz von Knoten der Fall, in der jeder nur eine ausgehende Kante hat und dadurch keine anderen Abläufe möglich sind. Ein Knoten übernimmt dabei einfach die Sequenzzugehörigkeit seines Vorgängers. Dies trifft in Abbildung 5.5 beispielsweise auf die Knoten $n_8..n_{11}$ zu. Weiterhin sind vor allem Knoten interessant, die mehrere ausgehende Kanten haben. Denn die Implementierung alternativer Implementierungskomponenten wird durch eine Aufspaltung des Kontrollflusses erreicht. Allerdings stellt nicht jede Verzweigung eine relevante Variabilität dar. Denn eine `IF`-Anweisung kann ebenso Teil der Implementierung eines Merkmals sein, sodass sowohl der `THEN`- als auch der `ELSE`-Zweig keine eigenständigen Merkmale darstellen. In diesem Fall würde der Kontrollfluss von einer Hilfsvariablen abhängen, wie bereits in Abschnitt 5.5.4 erläutert. Es deuten also nur solche Knoten auf verschiedene Merkmale hin, die mehrere ausgehende Kanten haben und an de-

nen ein p-use-Zugriff auf eine Merkmal- oder Interaktionsvariable erfolgt. Dies ist bei dem Knoten n_2 der Fall. In Knoten n_4 erfolgt ebenfalls ein p-use-Zugriff, allerdings auf eine Hilfsvariable. Er führt somit nicht zur Entstehung zweier neuer Sequenzen. Im Folgenden sind die Regeln noch einmal explizit formuliert, nach denen die Knoten einer Sequenz zugeordnet werden:

1. Ein Knoten wird keiner Sequenz zugeordnet, wenn ein p-use-Zugriff auf eine Merkmal- oder Interaktionsvariable erfolgt
2. Ein Knoten wird einer neuen Sequenz zugeordnet, wenn sein(e) Vorgänger keiner Sequenz zugeordnet sind
3. Ein Knoten wird einer neuen Sequenz zugeordnet, wenn seine Vorgänger verschiedenen Sequenzen zugeordnet sind
4. In allen anderen Fällen übernimmt der Knoten die Sequenzzugehörigkeit seines beziehungsweise seiner Vorgänger

Diese Regeln wurden auf den Kontrollflussgraphen aus Abbildung 5.2 angewandt. Das Ergebnis ist links in Abbildung 5.5 dargestellt, wobei jede Sequenz durch eine eigene Farbe repräsentiert wird. Anschließend wurden im rechten Graphen der Abbildung gleichfarbige Knoten zu einem rechteckigen Knoten zusammengefasst. Dieser stellt die jeweilige Sequenz dar. Die weiteren Schritte der Variabilitätsextraktion setzen einen Kontrollflussgraphen voraus, der auf die eben beschriebene Art verändert wurde. In Abbildung 5.5 konnten genau vier Sequenzen identifiziert werden. Es konnte also ermittelt werden, dass der Quelltext über vier Implementierungskomponenten verfügt. Die Beziehungen zwischen ihnen sind jedoch noch unbekannt. Bevor im folgenden Abschnitt gezeigt wird, wie auch diese extrahiert werden können, soll zuvor noch auf die Grenzen der Merkmalerkennung eingegangen werden.

Die eben beschriebene Methode orientiert sich ausschließlich an den Implementierungskomponenten des Lösungsraums bei der Definition von Merkmalen. Hierdurch entstehen andere Merkmale als im Rahmen der Domänenanalyse, da diese den Problemraum modelliert. Beispielsweise kann ein Merkmalmodell über zwei oder mehr obligatorische Merkmale verfügen, wie „Bildschirm“ und „Akku“. Da sie stets Bestandteil der Konfiguration sind, hat die gleichförmige Behandlung in einem Verarbeitungsschritt keine Auswirkung auf die generierbaren Produkte. Die Aggregation beider Merkmale ist semantisch aber mindestens zweifelhaft. So ließe sich schwer begründen, weshalb die Smartphone-Bestandteile „Bildschirm“ und „Akku“ ein gemeinsamen Merkmal darstellen sollten. Diese semantischen Unterschiede werden bei der Variabilitätsextraktion jedoch nicht festgestellt. Beispielsweise werden in Abbildung 5.5 die Anweisungen n_{13} und n_{14} zusammengefasst, selbst wenn diese keinen semantischen Zusammenhang aufweisen sollten. Es werden somit zu wenig Merkmale extrahiert. Vorstellbar ist auch der umgekehrte Fall, in dem die Implementierung eines Merkmals sich auf mehrere getrennte Quelltextabschnitte verteilt. Der in diesem Abschnitt beschriebene Algorithmus würde dann mehr als nur eine Sequenz identifizieren. Allerdings kann dies

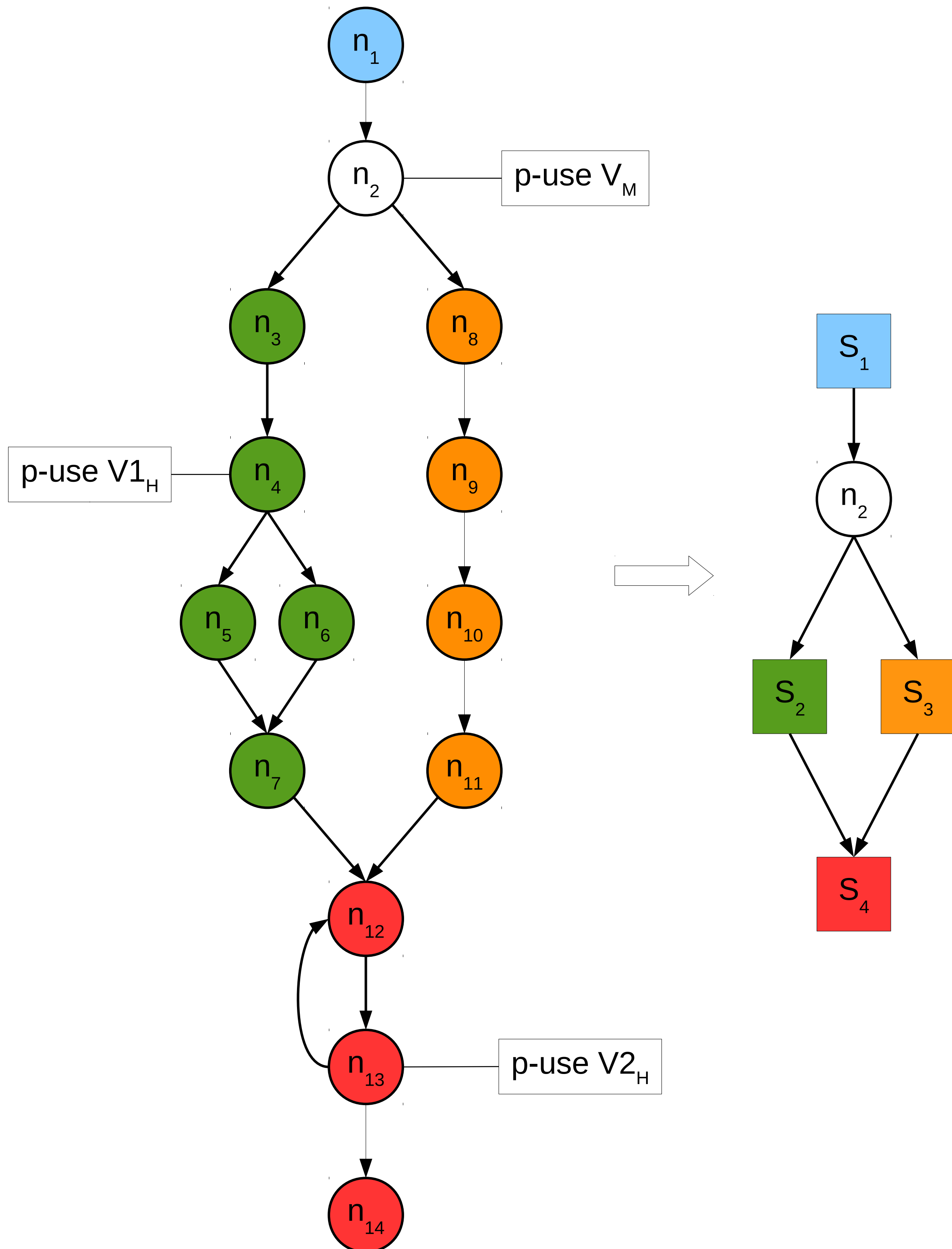

Abbildung 5.5: Kontrollflussgraph bevor (l.) und nachdem (r.) seine Knoten zu Sequenzen zusammengefasst wurden

in einigen Fällen durch die Analyse der Variablen entdeckt werden, wie unter anderem im nächsten Abschnitt gezeigt werden soll.

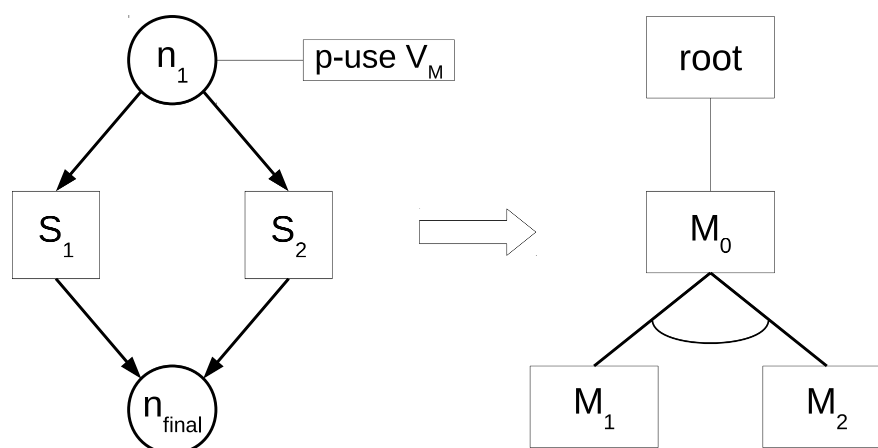

Abbildung 5.6: Transformation zweier alternativer Sequenzen in ein Merkmalmodell

5.7 Erkennen von Merkmalkardinalitäten

Im vorangegangenen Abschnitt wurde beschrieben, wie die Knoten des Datenflussgraphen in mehrere Sequenzen eingeteilt werden können. Nun soll untersucht werden, unter welchen Umständen sie Merkmalimplementierungen entsprechen und über welche Kardinalitäten die zugehörigen Merkmale verfügen. Dabei werden sowohl Einzelmerkmale als auch Merkmalgruppen berücksichtigt. Auf Cross-tree-Constraints wird aufgrund einiger Besonderheiten erst im nächsten Abschnitt eingegangen.

Sequenzen, die in jedem möglichen Pfad des Datenflussgraphs enthalten sind, entsprechen obligatorischen Merkmalen. Da im vorangegangenen Schritt bereits Knoten auf Grundlage der Variablenkategorisierung zusammengefasst wurden, enthält der Datenflussgraph nur noch Sequenzen und Verzweigungsknoten, in denen auf Merkmal- oder Interaktionsvariablen zugegriffen wird. In diesem Abschnitt ist nur erstgenannter Fall von Bedeutung. Jede noch vorhandene Verzweigung stellt also eine Entscheidung zwischen Sequenzen dar. Diese Situation ist in Abbildung 5.6 dargestellt. Der Graph wurde in eine XOR-Merkmalgruppe transformiert, da nur genau ein Merkmal Bestandteil des Produkts sein kann. Zur besseren Übersicht wird dabei ein abstraktes Merkmal M_0 erzeugt, dass über zwei konkrete Untermerkmale verfügt. Dies gilt analog für Verzweigungen mit mehr als zwei Sequenzen.

Abbildung 5.7 zeigt einen ähnlichen Graphen, allerdings fehlt die Sequenz M_2. Dieses Szenario lässt sich beispielsweise durch eine `IF`-Anweisung erreichen, die über keine `ELSE` -Anweisung verfügt. Wenn nur eine Sequenz vorliegt, die nur unter Umständen ausgeführt wird, entspricht dies exakt der Definition eines optionalen Einzelmerkmals, also einer Kardinalität $[0..1]$.

Komplizierter gestaltet sich das Erkennen von Merkmalgruppen, in denen mehr als ein Merkmal ausgewählt werden darf. Dies soll im Folgenden am Beispiel einer OR-Beziehung zweier Merkmale gezeigt werden. Aufgrund der möglichen Mehrfachauswahl kann die Implementierung nicht durch eine Verzweigung des Kontrollflusses erfolgen. Stattdessen müssen die Merkmale sequentiell implementiert werden, das heißt für jedes Merkmal muss eine eigene Verzweigung geschaffen werden. Dies ist in Abbildung 5.8 dargestellt. Da zunächst

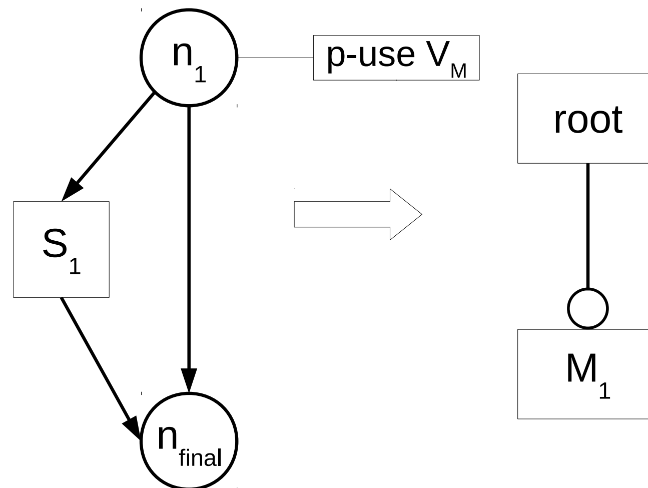

Abbildung 5.7: Transformation einer optionalen Sequenz in ein Merkmalmodell

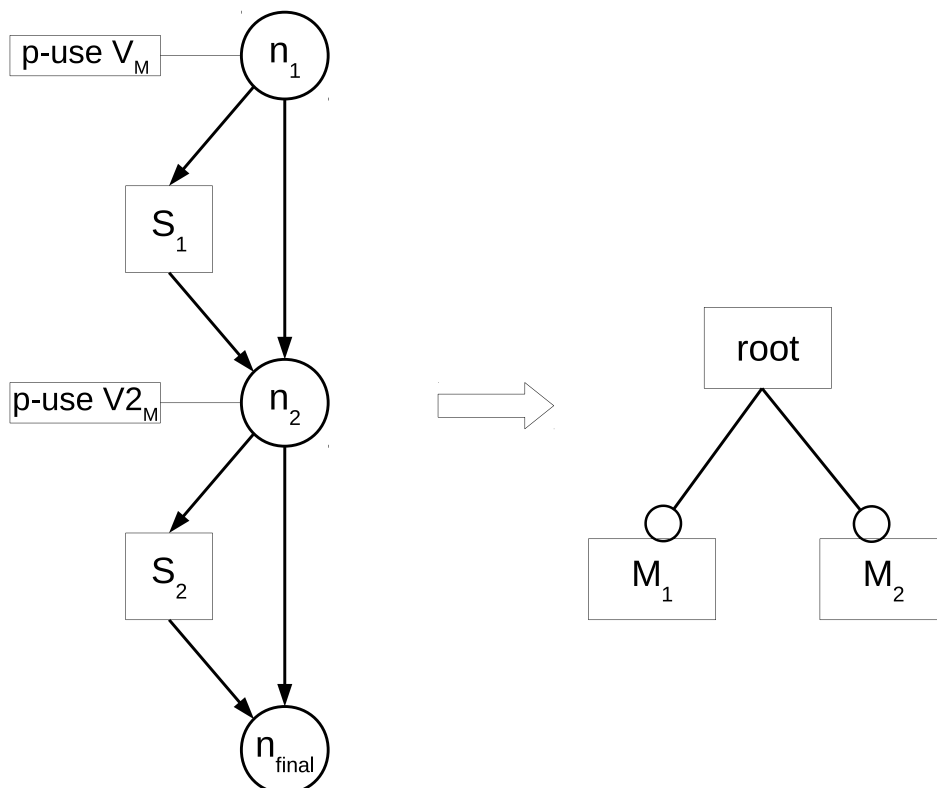

Abbildung 5.8: Transformation zweier optionaler Sequenzen in ein Merkmalmodell

keine weitere Abhängigkeiten zwischen beiden Merkmalen bekannt sind, würde dies entsprechend der zuvor beschriebenen Regel zur Modellierung optionaler Merkmale führen. Dies liegt daran, dass die Eigenschaft, dass mindestens eines der Merkmale selektiert sein muss, nicht extrahiert wurde. Hierauf wird erst im nächsten Abschnitt eingegangen, da es sich um eine Merkmalinteraktion handelt.

Abschließend soll das Erkennen hierarchischer Strukturen beschrieben werden. Diese sind im Kontrollfluss in Form einer doppelten Verzweigung sichtbar, wie in Abbildung 5.9 dargestellt. Ein derartiger Graph kann unter anderem durch eine Schachtelung von `IF`-Abfragen entstehen. Im Beispielgraphen können die Sequenzen M_2 und M_3 nur ausgeführt werden,

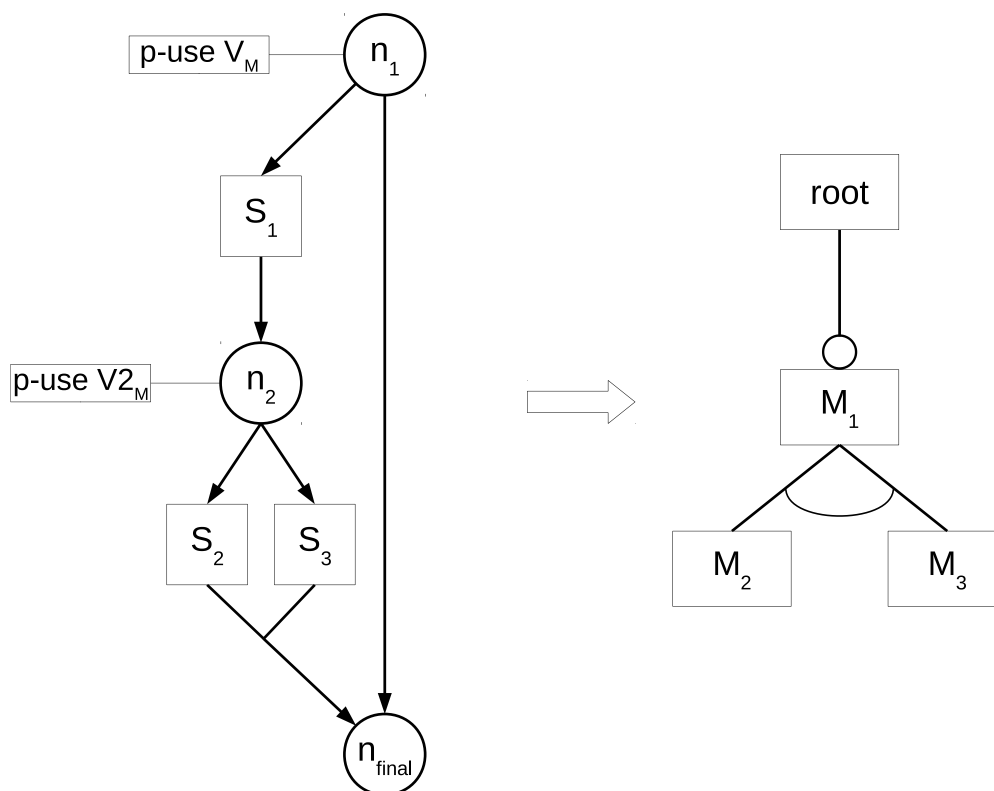

Abbildung 5.9: Transformation einer Mehrfachverzweigung in ein Merkmalmodell

wenn zuvor M_1 zur Ausführung kam. Dies trifft ebenso auf die Anordnung der Merkmale zu, die ebenfalls in Abbildung 5.9 gezeigt wird. Es ist jedoch auch möglich, auf die Hierarchie zu verzichten und stattdessen zwei Constraints `M_2 requires M_1` und `M_3 requires M_1` zu definieren. Beide Varianten sind äquivalent. Es soll allerdings folgender Grundsatz gelten: Ergibt sich eine derartige Beziehung zwischen zwei Sequenzen aus der Verzweigung des Kontrollflussgraphen, wie in Abbildung 5.9, so sollen die Merkmale hierarchisch angeordnet werden. Ergibt sich die Beziehung durch die im nachfolgenden Abschnitt vorgestellten Konstellationen, so sollen Constraints verwendet werden. Bei Cross-tree-Constraints handelt es sich um Abhängigkeiten zwischen Merkmalen unterschiedlicher Äste des Merkmalmodells. Eine geschachtelte `IF`-Abfrage stellt jedoch einen direkten Zusammenhang zwischen den Sequenzen beziehungsweise Merkmalen her, sodass eine hierarchische Anordnung dem besser Rechnung trägt. Das hier gezeigte Beispiel ist analog auf andere Kombinationen hierarchischer Merkmalanordnungen anwendbar.

5.8 Erkennen von Merkmalinteraktionen

Im vorangegangenen Abschnitt wurde die Erkennung von Merkmalkardinalitäten beschrieben. Hierzu wurden die Knoten näher untersucht, in denen ein p-use-Zugriff auf Merkmalvariablen erfolgt. Im Folgenden sollen Merkmalinteraktionen im Graphen identifiziert und extrahiert werden. Zu diesem Zweck sind hauptsächlich p-use-Zugriffe auf Interaktionsvariablen von Interesse.

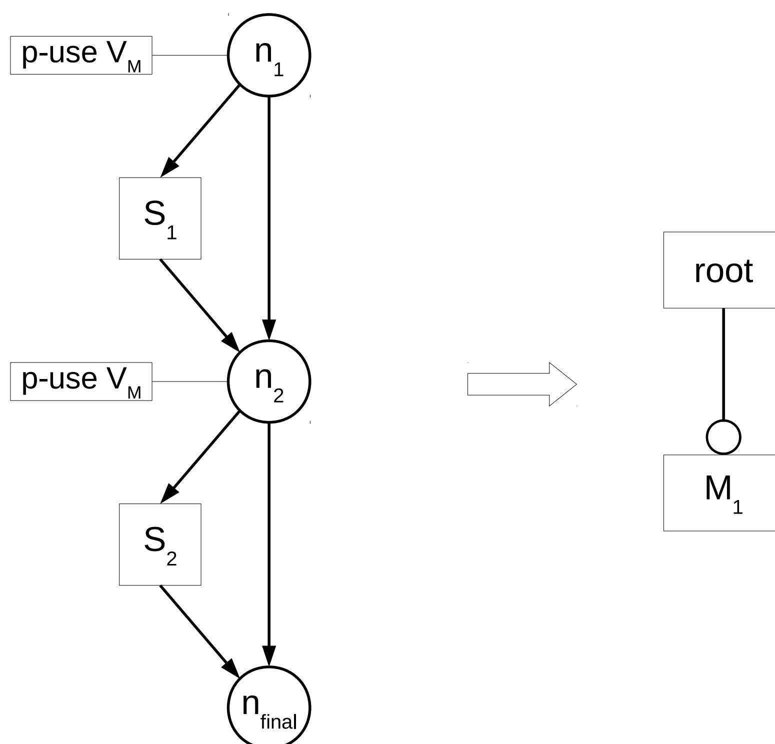

Abbildung 5.10: Transformation eines delokalisierten Merkmals

In einem Fall jedoch spielen auch Merkmalvariablen eine Rolle. Nämlich dann, wenn zwei unterschiedliche Sequenzen dasselbe Merkmal repräsentieren, es sich also um ein delokalisiertes Merkmal handelt. Abbildung 5.10 zeigt ein solches Szenario. Die beiden Sequenzen S_1 und S_2 werden zu einem Merkmal M_1 zusammengefasst. Hierzu müssen zwei Bedingungen erfüllt sein: An den beiden Verzweigungen n_1 und n_2 muss ein identischer p-use-Zugriff erfolgen. Zudem darf zwischen beiden Lesezugriffen kein Schreibzugriff auf die entsprechenden Variablen erfolgen. Nur dann ist sichergestellt, dass beide Sequenzen zur selben Merkmalimplementierung gehören.

Als nächstes werden die Merkmalinteraktionen im engeren Sinne betrachtet. Hierfür sind Knoten relevant, in denen ein p-use-Zugriff auf Interaktionsvariablen erfolgt. Zum Beispiel kann der weitere Kontrollfluss von der Evaluierung des Terms von $M_1 \wedge M_2$ abhängen, der sich in ADS durch eine Entscheidungstabelle implementieren lässt. Es wird eine Beziehung zwischen den beiden Merkmalen M_1 und M_2 ausgedrückt. Bevor sie in die Wissensbasis übernommen werden kann, muss geklärt werden, ob es sich um eine Merkmalinteraktion oder um eine Cross-tree-Constraint handelt. Der Unterschied äußert sich darin, dass im zweite Fall der Kontrollfluss endet, sofern die Bedingungen erfüllt ist. Im Fall der Bedingung $M_1 \wedge M_2$ muss eine entsprechende excludes-Beziehung modelliert werden. Eine requires-Abhängigkeit entspricht dem Term $M_1 \wedge \neg M_2$, der ebenfalls durch eine Entscheidungstabelle realisiert werden kann. Die Negation einer Variable ergibt sich aus dem Bedingungsanzeiger N, wie im nächsten Abschnitt gezeigt wird. Sofern der Kontrollfluss jedoch nicht abgebrochen wird, handelt es sich um eine Merkmalinteraktion. In diesem Fall kann der Zu-

```
1 .DTSTART
2 .DT-01.EQ.1,     N
3 .DT-02.EQ.1,     N
4 .DT              X,
5 .RC-8
6 .QUIT
7 .DTEND
```

Listing 5.6: Implementierung einer OR-Constraint zwischen Merkmalvariablen

sammenhang beider Merkmale, also beispielsweise $M_1 \wedge M_2$ ohne weitere Änderungen in die Wissensbasis übernommen werden.

Damit kann nun auch eine vollständige Erkennung einer OR-Beziehung erfolgen. Im vorangegangenen Abschnitt wurden nur die Merkmalvariablen untersucht. Daher konnte keine Beziehung zwischen den optionalen Merkmalen ermittelt werden, da hierzu Interaktionsvariablen erforderlich sind. Soll also ausgedrückt werden, dass von zwei optionalen Merkmalen mindestens eins gewählt werden muss, so entspricht dies dem Term $\neg M_1 \wedge \neg M_2$. Listing 5.6 zeigt eine Entscheidungstabelle, in der genau diese Beziehung zwischen den beiden Merkmalvariablen `01` und `02` existiert. Durch die Negation der beiden Bedingung durch den Bedingungsanzeiger N wird nicht die Merkmalinteraktion $M_1 \wedge M_2$ sondern $\neg M_1 \wedge \neg M_2$ extrahiert. Sofern die Regel erfüllt ist, muss der Rückgabewert 8 gesetzt und der Kontrollfluss abgebrochen werden, wie in Listing 5.6 geschehen.

Bisher wurden die verschiedenen Varianten aufgeführt, die im Datenflussgraph auftreten können und es wurde gezeigt, wie sie in ein Merkmalmodell überführt werden können. Dabei kann der Fall auftreten, dass eine Variable an verschiedenen Stellen im Quelltext sich scheinbar widersprechende Variabilitätsinformationen enthält, wie in Abbildung 5.11 beispielhaft gezeigt. Die Anordnung der Knoten n_1 und S_1 lässt dabei auf ein optionales Merkmal schließen. Unter der Annahme, dass es sich bei Merkmal 1 um ein delokalisiertes Merkmal handelt, dass sich aus den Sequenzen S_1 und S_2 zusammensetzt, ergeben sich jedoch bei Betrachtung der restlichen Knoten zusätzliche Informationen. Es wird deutlich, dass M_1 ein Element einer XOR-Merkmalgruppe ist. Tritt ein solcher Fall auf, muss das Merkmalmodell entsprechend angepasst werden, sofern dies auf einfache Weise realisierbar ist. Andernfalls kann die zusätzliche Information als aussagenlogische Formel in die Wissensbasis übernommen werden. Die graphischen Elemente müssen somit nicht angepasst werden, allerdings führt eine umfangreiche Wissensbasis zu einer geringeren Lesbarkeit. In beiden Fällen werden die im Quelltext verteilten Variabilitätsinformationen korrekt in das Merkmalmodell übernommen.

5.9 Benennung von Merkmalen

Merkmalmodelle die externe Variabilität modellieren, bedienen sich auch des Vokabulars des Problemraums. Bei der Variabilitätsextraktion liegen jedoch ausschließlich Artefakte vor, die dem Lösungsraum zuzuordnen sind. Es ist daher schwierig, anhand der gegebenen Informa-

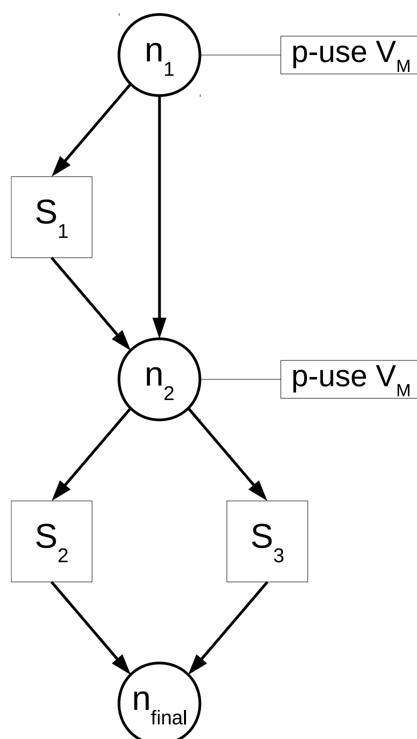

Abbildung 5.11: Delokalisiertes Merkmal mit unterschiedlichen Variabilitätsinformationen

tionen die Namen von Merkmalen zu identifizieren, wie sie im Rahmen der Variabilitätsmodellierung des Entwicklungsprozesses verwendet worden wären. Bei der Variabilitätsextraktion von Präprozessoranweisungen wurden die Makronamen ohne Änderung direkt als Merkmalname verwendet. Dies setzt jedoch voraus, das ein Merkmal immer durch eine Variable repräsentiert wird, wovon nicht ausgegangen werden kann. Zudem besteht das Problem, dass es sich bei ADS um keine Programmiersprache handelt, die als *sprechend* bezeichnet werden könnte. Sowohl die Schlüsselwörter als auch die Variablen bestehen meist nur aus zwei oder drei Zeichen. Ohne Kontext wird es in den meisten Fällen nicht möglich sein, diese korrekt zu interpretieren. Die Benennung eines Merkmals „CF“ stellt somit keinerlei Mehrwert dar.

Aus diesem Grund wurde sich dafür entschieden, eindeutige IDs zu verwenden. Da ihr Name keine Semantik trägt, lassen sie sich auf einfache Weise automatisch erzeugen. Ihre Aufgabe liegt in der Zuordnung zwischen den Elementen des AST und den Merkmalen. Beispielsweise erhält ein Merkmal die ID 1236. Es gibt nun ein oder mehrere ADS-Anweisungen, welche die Implementierung dieses Merkmals darstellen. Sie sind im AST durch XML-Elemente repräsentiert. Jedes dieser Elemente wird nun mit einem ID-Attribut versehen, das den Wert 123 enthält.

Auf diese Weise wird das manuelle Sichten des Quellcodes leichter, da einzelne Codestellen nun direkt Merkmalen zugeordnet werden können. Der Nachteil dieses Ansatzes liegt darin, dass das Merkmalmodell weiterhin nicht unmittelbar zum besseren Verständnis der Domäne beitragen kann. Es ist nicht ersichtlich, ob es sich bei einem optionalen Merkmal 123 um die Kamera oder den Akku des Smartphones oder um etwas gänzlich anderes handelt. Zumindest im Falle der ADS-Makros kann die Variabilitätsextraktion diese Informatio-

nen nicht aus dem Quelltext rekonstruieren. Es wäre zu untersuchen, ob es realisierbar ist, diese Informationen aus anderen Artefakten automatisch zu extrahieren. Eine Möglichkeit stellen eventuell die Konfigurationsdateien dar. Da diese jedoch nicht zur Verfügung stehen und sich die Arbeit auf die Variabilitätsextraktion des Quelltexts beschränken soll, ist hier die hilfsweise Beschränkung auf die IDs gegeben.

6 Anwendung

In diesem Kapitel soll gezeigt werden, wie die gewonnenen Erkenntnisse zur Variabilitätsextraktion angewandt werden können. Daher wird im ersten Abschnitt darauf eingegangen, wie eine empirische Untersuchung aussehen könnte. Die Grundlage bildet in jedem Fall eine Implementierung des Extraktionsprozesses. Daher wird in Abschnitt 6.2 der angefertigte Prototyp vorgestellt. Im Anschluss erfolgt eine beispielhafte Evaluation einiger Makros, die jedoch keine empirische Überprüfung ersetzen soll.

6.1 Empirische Überprüfung

Bei der in Abschnitt 5.5.4 vorgestellten Kategorisierung von Variablen handelt es sich bislang lediglich um theoretische Überlegungen. Diese setzt sich aus zwei Teilen zusammen: Zum einen aus der Kategorisierung an sich, zum anderen aus den Zuordnungsregeln. Die Richtigkeit beider Dimensionen kann jedoch nicht getrennt voneinander geprüft werden. Die Qualität der einzelnen Kategorien kann nicht beurteilt werden ohne eine entsprechende Zuteilung der Variablen vorzunehmen. Ein formaler Beweis müsste zeigen, dass Variablen, die anhand der aufgestellten Regeln zugeordnet werden, tatsächlich die behauptete Bedeutung für das Programm besitzen. Dies ist allerdings nicht möglich, da die Bedeutung der Variablen gegenwärtig nicht bekannt ist. Stattdessen muss die Kategorisierung einer kritischen empirischen Überprüfung unterzogen werden. Die Eignung kann dabei nur anhand des Mehrwerts für die Variabilitätsextraktion beurteilt werden.

Beispielsweise könnten mithilfe der vorgestellten Kategorisierung Merkmalmodelle erzeugt werden. Die im Rahmen dieser Arbeit erfolgte Implementierung in Form eines Prototyps kann hierfür die Grundlage bilden. Anschließend müssen Domänenexperten beurteilen, ob die identifizierten Merkmale und Interaktionen dem der Implementierung zugrunde liegenden Modellen entsprechen. Wenn ja, so ist dies ein Indiz für die Richtigkeit des Ansatzes. Im umgekehrten Fall jedoch sind entweder die Regeln oder die komplette Kategorisierung in Frage zu stellen. Sofern nur ein Teil fehlerhaft ist, ist es naheliegend, dass die Zuordnungsregeln verfeinert werden müssen. In einem iterativen Prozess erfolgt dann eine sukzessive Spezialisierung der Regeln, die eine immer feingliedrigere Zuordnung ermöglichen.

Die Motivation zur Variabilitätsextraktion am Beispiel der ADS-Generatoren ist eine fehlende Dokumentation. Daher ist nicht bekannt, welche Bedeutung den einzelnen Variablen zukommt oder welche Merkmale und Merkmalinteraktionen möglicherweise vorhanden sind. Werden nun automatisiert Merkmalmodelle aus den Makros extrahiert, kann nicht überprüft werden, ob die gewonnenen Informationen den ursprünglichen Intentionen entsprechen. Mithilfe des entwickelten Prototyps kann allenfalls bestätigt werden, dass Variablen existieren, die den entworfenen Regeln entsprechen. Es kann jedoch nicht gesagt werden, welche Bedeutung diese Variablen besitzen. Aus diesen Gründen kann die vorliegende Arbeit keine empirische Überprüfung der Variablenkategorisierung leisten.

6.2 Implementierung

Im Rahmen der vorliegenden Arbeit sollen die gefundenen Erkenntnisse auch praktisch angewandt und überprüft werden. Zu diesem Zweck wurde ein Prototyp entwickelt, der Variabilitätsinformationen aus ADS-Programmen extrahieren und in Form von Merkmalmodellen darstellen kann.

6.2.1 Funktionsumfang

Der Prototyp soll ADS-Makros einlesen, verarbeiten und anschließend die extrahierten Variabilitätsinformationen in Form eines Merkmalmodells darstellen. Die Verarbeitung erfolgt vollautomatisch. Um die Software im Rahmen dieser Arbeit umsetzen können, sollen bei der Erstellung des Datenflussgraphen nur die in Abschnitt 4.3 vorgestellten ADS-Befehle berücksichtigt, Das heißt `IF-ELSE`-Strukturen, Entscheidungstabellen, Schreibzugriffe via `SET` sowie `QUIT`-Anweisungen. Die Datenflussgraphen, welche durch diese Befehle möglich sind, können alle in den Abschnitten 5.6 – 5.8 vorgestellten Muster enthalten. Der Prototyp ist somit in der Lage, alle dort beschriebenen Merkmalarten und -beziehungen zu erkennen und in einem Merkmalmodell abzubilden. Zur graphischen Darstellung wird auf eine bestehende Software-Lösung zurückgegriffen. Dabei handelt es sich um *FeatureIDE*, ein Plugin für die Entwicklungsumgebung *Eclipse*, auf das im folgenden Abschnitt eingegangen wird.

6.2.2 FeatureIDE

FeatureIDE ist eine Umgebung zur Entwicklung von SPLs. Sie umfasst viele Teilfunktionalitäten, die auch getrennt voneinander genutzt werden können. Beispielsweise bietet FeatureIDE einen *Feature Model Editor*, einen Konfigurationseditor und Unterstützung für aspektorientierte und generative Entwicklung in mehreren Programmiersprachen (vgl. [Thüm et al. 2013]). Es unterstützt damit sowohl die Variabilitätsmodellierung als auch die Implementierung der SPL sowie den Spezifikationsprozess. Für die prototypische Implementierung wird jedoch nur der Feature Model Editor benötigt, der im Folgenden näher vorgestellt wird.

Er ermöglicht sowohl eine graphische als auch eine textuelle Repräsentation von Merkmalmodellen, allerdings nur in ihrer ursprünglichen Variante. Es ist somit nicht möglich, Kardinalitäten oder Wahrscheinlichkeitsangaben anzugeben. Dieser Einschränkung unterliegen auch alle gefundenen Alternativen wie der *Feature Diagram Editor* (vgl. [Klatt 2013]). Allerdings verfügt nur FeatureIDE über die erforderlichen Möglichkeiten, um Merkmalinteraktionen auszudrücken. Die Angabe erfolgt dabei analog zu der Notation, die in der vorliegenden Arbeit verwendet wird. Es wird eine Reihe logische Operatoren zur Verfügung gestellt, die sich zu vielfältigen Constraints kombinieren lassen. Zudem verfügt FeatureIDE über einen Layout-Algorithmus, der die einzelnen Elemente nach bestimmten Kriterien anordnet.

Weiterhin verfügt die Software über einen integrierten *Satisfiability-Solver (SAT-Solver)*. Darunter werden Programme verstanden, die ermitteln, ob eine aussagenlogische Formel erfüllbar ist oder nicht. Beispielsweise kann hierdurch festgestellt werden, ob ein Merkmal

nicht selektierbar ist, es wird dann als *totes Merkmal* bezeichnet. Dies kann im einfachsten Fall dadurch geschehen, indem eine excludes-Beziehung zwischen einem obligatorischen und diesem Merkmal existiert. Des Weiteren können so verstecke Merkmalinteraktionen entdeckt werden, die weder in der Formelmenge noch im Quelltext explizit vorhanden waren.

6.2.3 Funktionsweise

In diesem Abschnitt sollen die einzelnen Verarbeitungsschritte des Prototyps kurz vorgestellt werden. Auf konkrete Implementierungsdetails wird dabei verzichtet. Zunächst wird intern aus dem eingelesenen Makro ein Datenflussgraph aufgebaut. In diesem werden alle relevanten Informationen gespeichert. Dies umfasst Lese- und Schreibzugriffe sowie die Kategorie der Variablen. Weiterhin werden auch die Verknüpfungen zwischen Variablen gespeichert, wie sie sich beispielsweise aus einer Entscheidungstabelle ergeben. Auch anonyme Variablen, die beispielsweise aus Entscheidungstabellen gewonnen werden können, Anschließend wird ausschließlich mit diesem Graphen weitergearbeitet, weitere Informationen aus dem Makro sind nicht erforderlich. Auf diese Weise können Adapter geschrieben werden, die auch aus anderen Programmiersprachen einen solchen Datenflussgraphen erzeugen. Die weiteren Verarbeitungsschritte können dann ohne weiteren Transformationsaufwand ausgeführt werden.

Dabei werden zusätzlich Angaben zur Kategorie der Variablen gespeichert. Anschließend wird der Datenflussgraph in ein Merkmalmodell überführt. Dabei handelt es sich noch nicht um eine graphische Darstellung, sonder ebenfalls nur um eine interne Speicherung auf einem weiteren Graphen. Erst danach erfolgt die Serialisierung dieses internen Merkmalmodells. Derzeit kann nur das Format des FeatureIDE-Editors geschrieben werden. Auch hier kann das Programm durch weitere Adapter ergänzt werden, um auch andere Ausgabeformate zu unterstützen. Weiterhin ist die Ausgabe des Datenflussgraphen sowie der Variablenkategorisierung in Textform optional möglich.

6.3 Evaluation

In diesem Abschnitt soll die beispielhafte Auswertung eines Makros gezeigt werden. Aufgrund der Limitierungen des Prototyps ist es nicht möglich, eines der vorliegenden produktiv genutzten Makros auszuwerten. ADS kennt über ein Dutzend Möglichkeiten des Schreibzugriffs auf Variablen. Wird einer davon bei der Variabilitätsextraktion nicht berücksichtigt, kann nicht garantiert werden, dass beispielsweise die folgende Erkennung delokalisierter Merkmale korrekt ist. Daher soll das in Listing 6.1 gezeigte Makro verwendet werden.

Der Quelltext wird in einen AST umgewandelt und anschließend dem Prototypen übergeben. Dieser erzeugt nun den Datenflussgraphen, das Merkmalmodell (Abbildung 6.1) sowie eine Liste, die angibt, welche Anweisungen des Makros zur Implementierung welches Merkmals gehören (Tabelle 6.1). Das Merkmalmodell besteht aus insgesamt acht Merkmalen sowie drei Constraints. Das Wurzelelement sowie die XOR-Merkmalgruppe `XOR 0` sind durch

```
 1  Merkmal 1
 2 .IF-02.EQ.1
 3  Merkmal 2
 4 .IFELSE
 5  Merkmal 3
 6 .IFEND
 7 .IF-04.EQ.1
 8  Merkmal 4
 9 .IFEND
10 .IF-05.EQ.1
11  Merkmal 5
12 .IFEND
13 .DTSTART
14 .DT-02.EQ.1,    --Y
15 .DT-04.EQ.1,    -NY
16 .DT-05.EQ.1,    -N-
17 .DT     -XX,
18 .RC=8
19 .QUIT-MACRO
20 .DTEND
21 .IF-02.EQ.1
22  Merkmal 2
23 .IFEND
24 .SET-02=0
25 .IF-02.EQ.1
26  Merkmal 10
27 .IFEND
28 .DTSTART
29 .DT-04.EQ.1,    -Y
30 .DT-02.EQ.1,    -Y
31 .DT -X,
32  Merkmalinteraktion von 4 und 10
33 .DTEND
```

Listing 6.1: Makro mit mehreren Merkmalen und Merkmalinteraktionen

eine hellere Farbe gekennzeichnet, da es sich um abstrakte Merkmale handelt. Da sie keine Implementierung besitzen, sind ihnen in Tabelle 6.1 keinerlei Quelltextabschnitte zugeordnet.

Das Merkmalmodell hilft bei der Analyse des Quelltexts. Beispielsweise ist ersichtlich, dass es ein delokalisiertes Merkmal gibt. Die Implementierung des Merkmals 2 besteht aus den Zeilen 3 und 22 des Listings, denn beiden geht der identische p-use-Zugriff `.IF-02.EQ.1` voraus. Dies gilt ebenso für Zeile 21. Aufgrund des zuvor erfolgten Schreibzugriffs auf Variable `02` ist die nachfolgende Zeile jedoch kein Bestandteil der Implementierung des Merkmals 2. Stattdessen wird ein eigenes, optionales Merkmal 10 erzeugt.

Mit Hilfe der ersten Entscheidungstabelle werden zwei Constraints implementiert. Diese wurden wie beschrieben extrahiert und sind ebenfalls in Abbildung 6.1 in Form propositiona-

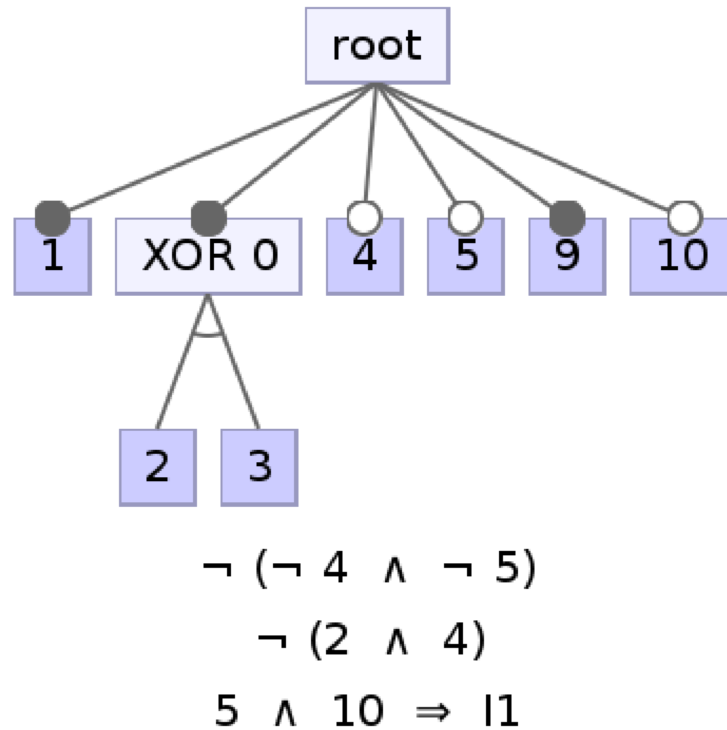

Abbildung 6.1: Aus Listing 6.1 erzeugtes Merkmalmodell

ler Formeln enthalten. Die erste Constraint $\neg(\neg 4 \wedge \neg 5)$ ist äquivalent zu $4 \vee 5$, es handelt sich also um eine OR-Beziehung zwischen beiden Merkmalen. Der Term $\neg 2 \ \wedge 4$ entspricht einer excludes-Beziehung, da die Merkmale 2 und 4 nicht zusammen selektiert werden dürfen. Die zweite Entscheidungstabelle beschreibt eine Merkmalinteraktion zwischen den Merkmalen 4 und 10. Da es sich um eine Abhängigkeit im Lösungsraum handelt, wird sie nicht als eigenständiges Merkmal modelliert. In Tabelle 4.1 ist aufgeführt, welche Codezeilen ausgeführt werden, wenn die entsprechende Merkmalkombination vorliegt.

Merkmal/Interaktion	Implementierung
Merkmal 1	Zeile 1
Merkmal 2	Zeile 3, 22
Merkmal 3	Zeile 5
Merkmal 4	Zeile 8
Merkmal 5	Zeile 11
Merkmal 9	Zeile 24
Merkmal 10	Zeile 26
Interaktion 1	Zeile 32

Tabelle 6.1: Zuordnungen von Code-Abschnitten zu Merkmalen und Merkmalinteraktionen

Auffällig ist weiterhin, dass Merkmal 4 in jeder der aufgeführten Constraints vorkommt. Es weist Abhängigkeiten zu der Hälfte aller übrigen konkreten Merkmale auf. Dies ist ein Indiz für eine verbesserungsfähige Definition der Merkmale. In einer weiterführenden manuellen Analyse kann nun das betroffene Merkmal näher untersucht werden, um mögliche Probleme zu finden.

Es wurde gezeigt, wie die extrahierten Artefakte zu einer verbesserten Analyse des Quelltexts beitragen können. Insbesondere delokalisierte können direkt identifiziert werden. Dies ist ohne Merkmalmodell mit einem sehr viel höherem Aufwand verbunden. Allerdings wird ebenfalls deutlich, dass die Variabilitätsextraktion nur Informationen über den Lösungsraum liefert. Um zu verstehen, welche Bedeutung die einzelnen Merkmale tragen, muss weiterhin eine manuelle Analyse des Quelltexts erfolgen.

7 Schlussbetrachtung

Die vorliegende Arbeit beschäftigte sich mit der Variabilitätsextraktion am Beispiel von ADS-Makros. Es konnte zum einen gezeigt werden, dass es sich dabei um Generatorsysteme handelt, die in der Lage sind, alle vorgestellten Formen der Variabilität abzubilden. Zum anderen können diese Systeme über Konfigurationswissen verfügen, das bei einer Modernisierung nicht verloren gehen soll. Aus diesem Grund umfasste die Variabilitätsextraktion neben der Merkmalerkennung auch das Erkennen von Merkmalinteraktionen und Cross-tree-Constraints. Die vorliegende Arbeit liefert einen Beitrag zu einer besseren Dokumentation der ADS-Makros. Für eine Modernisierung sind jedoch noch weitere Aspekte zu berücksichtigen, beispielsweise eine Architekturanalyse.

Die größte Herausforderung für die Variabilitätsextraktion besteht in der großen Anzahl an unterschiedlichen Fällen, die berücksichtigt werden müssen. Diese Vielfalt ergibt sich aus mehreren Gegebenheiten: Sie entsteht zum einen dadurch, dass jede Kontrollflussstruktur sowie jeder Lese- und Schreibzugriff untersucht werden muss. Zum anderen gibt es mehrere unterschiedliche Sprachmittel, um beispielsweise einen Schreibzugriff zu realisieren. Für eine vollständige Variabilitätsextraktion muss jede der daraus ableitbaren Kombinationen berücksichtigt werden.

Es wurde gezeigt, wie von konkreten ADS-Anweisungen abstrahiert werden kann, sodass ein sprachunabhängiger Datenflussgraph entsteht. Auf diese Weise konnte die Anzahl der zu untersuchenden Fälle deutlich reduziert werden. Des Weiteren ermöglicht die Arbeit auf dem Datenflussgraph einen sprachunabhängigen Extraktionsprozess. Dies vereinfacht die Übertragung des Vorgehens auf andere Generatorsysteme oder SPLs.

Bisherige Ansätze gehen davon aus, dass jeder untersuchte Befehl der Implementierung von Merkmalkardinalitäten dient. Dies stellt eine Einschränkung dar, die in der Praxis nicht immer erfüllt ist und zur Identifikation einer falschen Anzahl an Merkmalen führt. Daher wurde im Rahmen dieser Arbeit eine Unterteilung von Variablen in drei Kategorien erarbeitet, um diese Abweichung zu verringern. Merkmalvariablen dienen der Identifizierung von Merkmalen. Interaktionsvariablen beschrieben die Beziehungen zwischen Merkmalen, die mithilfe von Merkmalbeziehungen und Cross-tree-Constraints modelliert werden können. Um eine eindeutige Kategorisierung zu ermöglichen, wurden einige Regeln aufgestellt, die jeder Variablen einer Kategorie zuordnen. In weiterführenden Arbeiten muss jedoch eine empirische Überprüfung dieser Regeln erfolgen, um Ausnahmen zu entdecken und die Regeln gegebenenfalls daran anzupassen. Insbesondere sei hier auf die Verwendung von Vorgabewerten hingewiesen, die nur dann verwendet werden, wenn der Benutzer keine Auswahl trifft. Diese werden nach den aufgestellten Regeln als Hilfsvariablen klassifiziert, obwohl es sich um Merkmalvariablen handelt.

Auch wenn viele Variabilitätsinformationen aus dem Quelltext des Generators extrahiert werden können, führt eine automatische Variabilitätsextraktion nicht zu den selben Merk-

malmodellen wie manuell erstellte. Es kann nach wie vor nicht ausgeschlossen werden, dass verschiedene Codeabschnitte einem Merkmal zugeordnet werden, obwohl sie keinen semantischen Bezug zueinander haben. Zudem wurde deutlich, dass die generierten Merkmalmodelle helfen den Lösungsraum besser zu verstehen, aufgrund ihrer Nähe zum Quelltext allerdings kaum zu einem besseren Verständnis der Domäne beziehungsweise des Problemraums beitragen. Hierzu muss auf andere Methoden als die automatische Variabilitätsextraktion zurückgegriffen werden.

Weiterhin wurde ein Prototyp entwickelt, der eine Implementierung des beschriebenen Extraktionsprozesses darstellt. Aufgrund der bereits geschilderten Vielzahl an Möglichkeiten wurde sich auf eine eng umgrenzte Teilmenge der ADS-Anweisungen begrenzt, die dennoch die Extraktion verschiedener Merkmalarten und Merkmalinteraktionen ermöglicht. Um auch produktiv eingesetzte Makros analysieren zu können, muss der unterstützte Sprachumfang weiter ausgebaut werden. Anhand der Auswertung eines Beispielmakros wurde weiterhin gezeigt, wie die generierten Merkmalmodelle eingesetzt werden können und welchen Mehrwert sie für die Analyse des Quelltexts liefern.

In weiteren Arbeiten kann der beschriebene Extraktionsprozess auf mehrere Arten erweitert beziehungsweise verbessert werden. Erstens muss der vollständige ADS-Sprachumfang berücksichtigt werden. Darunter fallen einerseits alternative Implementierungsmöglichkeiten von Konfigurationswissen und andererseits in dieser Arbeit nicht berücksichtigte Konzepte wie Programmbausteine und späte Bindung. Des Weiteren muss untersucht werden, ob die Identifikation der Merkmale und Merkmalbeziehungen verbessert werden kann. Gegenwärtig ist deren Erkennung nicht exakt. Es ist möglich, dass verschiedene Merkmale fälschlicherweise gruppiert werden oder ein Zusammenhang zwischen delokalisierten Implementierungskomponenten nicht erkannt wird. Ein Ansatzpunkt können hierbei eventuell die Konfigurationen darstellen. Es ist zu untersuchen, ob sie zu einer besseren Merkmalerkennung beitragen können.

Literaturverzeichnis

[Acher 2011] Mathieu Acher. „Managing Multiple Feature Models: Foundations, Language, and Applications“. Diss. Nice und France: University of Nice Sophia Antipolis, 2011. URL: https://nyx.unice.fr/publis/acher:2011.pdf (besucht am 02. 12. 2013).

[Antkiewicz et al. 2009] Michael Antkiewicz, Thiago Tonelli Bartolomei und Krzysztof Czarnecki. „Fast extraction of high-quality framework-specific models from application code“. In: *Automated Software Engg.* 16.1 (März 2009), S. 101–144. DOI: 10.1007/s10515-008-0040-x.

[Balzert 1998] Helmut Balzert. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akad. Verl., 1998. ISBN: 9783827400659.

[Balzert 2011] Helmut Balzert. *Lehrbuch der Software-Technik: Entwurf, Implementierung, Installation und Betrieb*. 3. Aufl. Spektrum Akademischer Verlag, 2011. DOI: 10.1007/978-3-8274-2246-0.

[Batory et al. 2003] Don Batory, Jacob Neal Sarvela und Axel Rauschmayer. „Scaling stepwise refinement“. In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. Washington, DC und USA: IEEE Computer Society, 2003, S. 187–197. ISBN: 0-7695-1877-X. URL: http://dl.acm.org/citation.cfm?id=776816.776839.

[Bennicke/Rust 2004] Marcel Bennicke/Heinrich Rust. „Programmverstehen und statische Analysetechniknen im Kontext des Reverse Engingeering und der Qualitätssicherung“. In: ViSEK (2004).

[Bosch/Capilla 2012] Jan Bosch/Rafael Capilla. „Dynamic Variability in Software-Intensive Embedded System Families“. In: *Computer* 45.10 (2012), S. 28–35. DOI: 10.1109/MC.2012.287.

[Cardozo et al. 2011]. *Feature-Oriented Programming and Context-Oriented Programming: Comparing Paradigm Characteristics by Example Implementations*. 6th International Conference on Software Engineering Advances. 2011.

[Clements/Northrop 2007] Paul Clements/Linda Northrop. *Software Product Lines: Practices and Patterns*. 6. Aufl. SEI Series in Software Engineering. Boston: Addison-Wesley, 2007.

[Czarnecki 2013] Krzysztof Czarnecki. „Variability in Software: State of the Art and Future Directions“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von

Vittorio Cortellessa/Dániel Varró. Bd. 7793. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, S. 1–5. DOI: 10.1007/978-3-642-37057-1_1.

[Czarnecki/Eisenecker 2000] Krzysztof Czarnecki/Ulrich W. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[Czarnecki et al. 2012] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid und Andrzej Wasowski. „Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches“. In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS '12. New York, NY und USA: ACM, 2012, S. 173–182.

[Czarnecki et al. 2005a] Krzysztof Czarnecki, Simon Helsen und Ulrich W. Eisenecker. „Formalizing Cardinality-based Feature Models and their Specialization“. In: *Software Process: Improvement and Practice* 10.1 (2005), S. 7–29.

[Czarnecki et al. 2005b] Krzysztof Czarnecki, Simon Helsen und Ulrich W. Eisenecker. „Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models“. In: *Software Process: Improvement and Practice* 10.2 (2005), S. 143–169.

[Czarnecki/Pietroszek 2006] Krzysztof Czarnecki/Krzysztof Pietroszek. „Verifying feature-based model templates against well-formedness OCL constraints.“ In: *GPCE*. Hrsg. von Stan Jarzabek, Douglas C. Schmidt und Todd L. Veldhuizen. ACM, 2006, S. 211–220. DOI: 10.1145/1173706.1173738.

[Czarnecki et al. 2008] Krzysztof Czarnecki, Steven She und Andrzej Wasowski. „Sample Spaces and Feature Models: There and Back Again“. In: *Proceedings of the 2008 12th International Software Product Line Conference*. SPLC '08. Washington, DC und USA: IEEE Computer Society, 2008, S. 22–31.

[Deelstra et al. 2004] Sybren Deelstra, Marco Sinnema und Jan Bosch. „Experiences in Software Product Families: Problems and Issues During Product Derivation“. In: *Software Product Lines*. Hrsg. von RobertL. Nord. Bd. 3154. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, S. 165–182. DOI: 10.1007/978-3-540-28630-1_10.

[Deelstra et al. 2005] Sybren Deelstra, Marco Sinnema und Jan Bosch. „Product derivation in software product families: a case study“. In: *J. Syst. Softw.* 74.2 (Jan. 2005), S. 173–194. DOI: 10.1016/j.jss.2003.11.012.

[DSTG 2012a] Delta Software Technology GmbH. *ADS™ Band 1: Grundlagen*. 2012.

[DSTG 2012b] Delta Software Technology GmbH. *ADS™ Band 3: Programmlogik*. 2012.

[DSTG 2013] Delta Software Technology GmbH. *AmAVaG - Automatische Architektur- und Variabilitätsanalyse in Generatorsystemen*. 2013. URL: http://www.d-s-t-g.com/de/forschung/gp-partner/amavag.html (besucht am 09. 12. 2013).

[Egyed 2001] Alexander Egyed. „A scenario-driven approach to traceability". In: *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*. 2001, S. 123–132. DOI: 10.1109/ICSE.2001.919087.

[Ferber et al. 2002] Stefan Ferber, Jürgen Haag und Juha Savolainen. „Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line". In: *Proceedings of the Second International Conference on Software Product Lines*. SPLC 2. London, UK und UK: Springer-Verlag, 2002, S. 235–256. ISBN: 3-540-43985-4. URL: http://dl.acm.org/citation.cfm?id=645882.672250.

[Filho/Redmiles 2007] Roberto Silveira Silva Filho/David F. Redmiles. „Managing Feature Interaction by Documenting and Enforcing Dependencies in Software Product Lines." In: *ICFI*. Hrsg. von Lydie du Bousquet/Jean-Luc Richier. IOS Press, 2007, S. 33–48. ISBN: 978-1-58603-845-8.

[Gurp et al. 2001] Jilles van Gurp, Jan Bosch und Mikael Svahnberg. „On the Notion of Variability in Software Product Lines". In: *In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01*. IEEE Computer Society, 2001, S. 45–54.

[Kim et al. 2008] Chang Hwan Peter Kim, Christian Kästner und Don Batory. „On the modularity of feature interactions". In: *Proceedings of the 7th international conference on Generative programming and component engineering*. GPCE '08. New York, NY und USA: ACM, 2008, S. 23–34. DOI: 10.1145/1449913.1449919.

[Klatt 2013] Benjamin Klatt. *Feature Diagram Editor*. URL: http://wiki.eclipse.org/EMF_Feature_Model/Feature_Diagram_Editor (besucht am 27. 12. 2013).

[Loesch/Ploedereder 2007] Felix Loesch/Erhard Ploedereder. „Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations". In: *11th European Conference on Software Maintenance and Reengineering*. 2007, S. 159–169.

[Lozano 2011] Angela Lozano. „An Overview of Techniques for Detecting Software Variability Concepts in Source Code". In: *Proceedings of the 30th international conference on Advances in conceptual modeling: recent developments and new directions*. ER'11. Berlin und Heidelberg: Springer-Verlag, 2011, S. 141–150.

[Madhavji et al. 2006] Nazim H. Madhavji, Juan C. Fernández-Ramil und Dewayne E. Perry, Hrsg. *Software evolution and feedback : Theory and practice*. Includes bibliographical references and index. Chichester [u.a.]: Wiley, 2006, XXXV, 575 S. ISBN: 9780470871805.

[Ottmann/Widmayer 1996] Thomas Ottmann/Peter Widmayer. *Algorithmen und Datenstrukturen*. 3. Aufl. Spektrum Lehrbuch. Spektrum, 1996. ISBN: 978-3-8274-0110-6.

[Parra et al. 2010] Carlos Parra, Anthony Cleve, Xavier Blanc und Laurence Duchien. „Feature-Based Composition of Software Architectures“. In: *Software Architecture*. Hrsg. von MuhammadAli Babar/Ian Gorton. Bd. 6285. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, S. 230–245. DOI: 10.1007/978-3-642-15114-9_18.

[Piller 1997] Thomas Piller. „Individualität von der Stange: Mass Customization“. In: *Harvard Business Manager* 1997.8 (1997).

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

[She 2008] Steven She. „Feature Model Mining“. Masterarbeit. Waterloo: University of Waterloo, Aug. 2008. URL: http://hdl.handle.net/10012/3915 (besucht am 02. 12. 2013).

[She et al. 2012] Steven She, Krzysztof Czarnecki und Andrzej Wąsowski. „Usage Scenarios for Feature Model Synthesis“. In: *Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone*. VARY '12. Innsbruck, Austria: ACM, 2012, S. 15–20. DOI: 10.1145/2425415.2425419.

[Snelting 1996] Gregor Snelting. „Reengineering of Configurations Based on Mathematical Concept Analysis“. In: *ACM Transactions on Software Engineering and Methodology* 5.2 (Apr. 1996), S. 146–189. DOI: 10.1145/227607.227613.

[Soloway et al. 1988] Elliot Soloway, Robin Lampert, Stan Letovsky, David Littman und Jeannine Pinto. „Designing Documentation to Compensate for Delocalized Plans“. In: *Commun. ACM* 31.11 (Nov. 1988), S. 1259–1267. DOI: 10.1145/50087.50088.

[Thaker et al. 2007] Sahil Thaker, Don Batory, David Kitchin und William Cook. „Safe composition of product lines“. In: *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*. New York, NY, USA: ACM Press, 2007, S. 95–104. DOI: 10.1145/1289971.1289989.

[Thüm et al. 2013] Thomas Thüm et al. *FeatureIDE*. Universität Magdeburg. URL: wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/ (besucht am 25. 11. 2013).

[Weiss et al. 2005]. *Invocation Order Matters: Functional Feature Interactions of Web Services*. International Workshop on Engineering Service Compositions (WESC). IBM Research Report RC23821, 2005, S. 69–76.

[Würsch 2006] Michael Würsch. „Improving Abstract Syntax Tree based Source Code Change Detection“. Masterarbeit. University of Zurich, 2006.

[Yang et al. 2009] Yiming Yang, Xin Peng und Wenyun Zhao. „Domain Feature Model Recovery from Multiple Applications Using Data Access Semantics and Formal

Concept Analysis“. In: *Proceedings of the 2009 16th Working Conference on Reverse Engineering*. WCRE '09. Washington, DC und USA: IEEE Computer Society, 2009, S. 215–224. DOI: 10.1109/WCRE.2009.15.

Ehrenwörtliche Erklärung

Ich versichere, dass ich die Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Darüber hinaus versichere ich, dass die elektronische Version der Masterarbeit mit der gedruckten Version übereinstimmt.