
Emmy Noether-Forschungsgruppe Softwareproduktlinien
Dr.-Ing. Sven Apel

**Masterarbeit**
im Studiengang Informatik

# Typprüfung von Produktlinien in Fuji

30. April 2013

**Peter Lutz**
Bachgasse 4
93109 Wiesent
p_lutz@gmx.de

| | |
|---|---|
| Erstprüfer: | Dr.-Ing. Sven Apel |
| Zweitprüfer: | Prof. Christian Lengauer, Ph.D. |
| Betreuer: | Sergiy Kolesnikov |

## Abstract

Eine *Softwareproduktlinie* ist eine Menge von Softwareprodukten, die auf einer gemeinsamen Quelltextbasis aufbauen. Die Funktionalitäten dieser Quelltextbasis werden *Features* zugeordnet. Die einzelnen Produkte haben bestimmte Features gemeinsam, unterscheiden sich aber in anderen Features. Die Produkte werden anhand einer Auswahl an Features generiert.

Alle Produkte einer Softwareproduktlinie einzeln auf Typfehler zu überprüfen ist bei großen Produktlinien nicht mehr durchführbar. Schlimmstenfalls wächst die Anzahl der Produkte exponentiell mit der Anzahl der Features.

In der *feature-orientierten Programmierung* wird im *familienbasierten Ansatz* die Produktlinie als Ganzes anhand einer abstrakten Repräsentation der gesamten Codebasis und des Feature-Modells auf Typfehler überprüft. Stimmen Abhängigkeiten zwischen Features aufgrund von Zugriffen in der Implementierung nicht mit den Abhängigkeiten, die das Feature-Modell vorgibt, überein, so kann es passieren, dass bei der Komposition von laut Feature-Modell validen Produkten Zugriffe aus Features heraus in diesem Produkt kein Ziel finden, da benötigte Features nicht in die Komposition miteinbezogen wurden. Dies kann dadurch entstehen, dass benötigte Features im Feature-Modell fälschlicherweise als optional oder als unvereinbar mit ausgewählten Features gekennzeichnet sind, oder dadurch, dass benötigte Informationen über Subtyp-Beziehungen nur in fälschlicherweise als optional oder als unvereinbar mit ausgewählten Features gekennzeichneten Features bereitgestellt werden.

In dieser Arbeit wird für den Softwareproduktlinien-Compiler Fuji ein *familienbasierter Typchecker für feature-orientierte Softwareproduktlinien* entwickelt, der Zugriffe zwischen Features mit den Abhängigkeiten, die im Feature-Modell vorgegeben werden, vergleicht und für gefundene Typfehler eine detaillierte Fehlermeldung ausgibt.

Beim Vergleich der Performanz dieses entwickelten Typcheckers mit der Performanz der naiven, produktbasierten Typprüfung aller Produkte zeigte sich, dass der Typchecker bei kleinen Produktlinien mit wenigen Features unterlegen, bei größeren Produktlinien mit einfachem Feature-Modell und vielen Produkten aber um den Faktor 10 bis 20 schneller ist.

## Danksagung

Ich möchte mich bei meinem Betreuer Sergiy Kolesnikov und allen Korrekturlesern bedanken.

# Inhaltsverzeichnis

# 1 Einleitung

Unter einer *Softwareproduktlinie* versteht man eine Familie von Softwaresystemen, die auf einer gemeinsamen Quellcodebasis aufbauen. Dabei haben die einzelnen Systeme bestimmte Features gemeinsam, unterscheiden sich aber in anderen Features [CN02], [PBL05].

Diese Features stellen eine Konfigurationsmöglichkeit da, um die aus der Softwareproduktlinie erzeugten Produkte für einen konkreten Einsatz innerhalb eines Anwendungsbereichs anzupassen [Rhe+13].

Durch vorgeschriebene Variationsmechanismen, die für die gesamte Produktlinie gelten, werden die Features aus einer gemeinsamen Grundmenge für ein bestimmtes Produkt ausgewählt. Durch das Hinzufügen neuer Features kann diese Grundmenge erweitert werden, ohne bestehende Produkte verändern zu müssen. Das Erstellen eines neuen Produkts reduziert sich im Idealfall auf die Auswahl zueinanderpassender Features [CN02].

Diese Vorgehensweise bietet bezüglich der Wartbarkeit Vorteile gegenüber der sogenannten *clone and own*-Strategie, bei der bestehende Software kopiert und anschließend modifiziert wird, um die Software an bestimmte Anforderungen eines konkreten Verwendungszwecks anzupassen [DB07]. Wird ein Fehler in der gemeinsamen Codebasis entdeckt, so genügt es bei Softwareproduktlinien, die gemeinsame Codebasis zu überarbeiten, aus der die einzelnen Produkte erzeugt werden. Betrifft dieser Fehler aber eine Vielzahl von kopierten und modifizierten eigenständigen Softwaresystemen, so fällt deutlich höherer Wartungsaufwand an [DB07].

Es gibt verschiedene Methodiken, um eine Softwareproduktlinie zu erstellen [AK09]. Eine Methodik ist die *feature-orientierte Programmierung* (FOP). Dabei wird der Programmtext in Feature-Module gekapselt. Anhand eines Feature-Modells, das die Beziehungen und Abhängigkeiten zwischen Features angibt, wird eine gültige Auswahl dieser Module zu einem Produkt zusammengefügt [AK09].

Zur Erstellung von brauchbaren und vertrauenswürdigen Softwareproduktlinien sind automatische Analyseverfahren unabdingbar. Theoretisch könnten dazu bestehende Analysemethoden wie die Typprüfung, andere statische Analyseverfahren und formale Verifikation angewandt werden, indem jedes valide Produkt der Produktlinie einzeln erzeugt und auf Fehler überprüft wird (produktbasiertes Analyseverfahren). Allerdings wächst die Anzahl der möglichen Produkte gerade bei großen Softwareproduktlinien mit vielen unabhängigen Features im schlimmsten Fall exponentiell mit der Anzahl der Features, wodurch dieser naive Ansatz praktisch nicht durchführbar ist [Thü+12].

Aufbauend auf bestehenden Analyseverfahren für einzelne Softwaresysteme werden deshalb angepasste Verfahren für Produktlinien entwickelt und eingesetzt. Einen Überblick über diese Verfahren bietet [Thü+12].

Die *familienbasierte* Analyse für die feature-orientierte Programmierung wird anhand einer abstrakten Repräsentation der Codebasis, die durch alle Features gebildet wird, und anhand eines Feature-Modells, das alle gültigen Kombinationen dieser Features ausweist, durchgeführt [Thü+12].

Diese eigentliche Analyse übernimmt beispielsweise ein Typchecker.

Aufgabe eines Typcheckers ist es, sicherzustellen, dass das Typsystem einer Programmiersprache bei der Implementierung beachtet wird [Pie02]. So wird z.B. ein Fehler gemeldet, wenn ein Feld, auf das zugegriffen wird, nicht deklariert wurde. Bei Softwareproduktlinien ergeben sich durch das Zusammenfügen von Feature-Modulen weitere Fehlerquellen.

Bestehen Diskrepanzen zwischen den Abhängigkeiten, die sich durch die Implementierung der einzelnen Features ergeben, und den Abhängigkeiten, die im Feature-Modell vorgegeben werden, so kann es passieren, dass sich in den Features Referenzen auf Felder, Methoden und Typen in anderen Features befinden, die im Feature-Modell nicht berücksichtigt werden. Wird ein laut Feature-Modell valides Produkt erzeugt, kann es so passieren, dass Referenzen kein Ziel in diesem Produkt finden, da ein benötigtes Feature (mit dem benötigten Feld, der benötigten Methode oder dem benötigten Typ) nicht zum Produkt hinzukomponiert wurde. Dies führt zu einem Typfehler beim Kompilieren dieses Produkts.

Wenn diese Probleme nicht auf Tippfehler oder ähnliches zurückzuführen sind, können diese Probleme durch das Hinzufügen von geeigneten Abhängigkeiten zum Feature-Modell behoben werden. Bei der Auswahl eines Features mit einer Referenz auf ein anderes Feature wird somit sichergestellt, dass auch dieses benötigte, aber fälschlicherweise als optional gekennzeichnete Feature zwingend ausgewählt werden muss (wodurch die Anzahl an Variantsmöglichkeiten des Feature-Modells sinkt). Bei als unvereinbar mit einem ausgewählten Feature gekennzeichneten Features gestaltet sich die Auflösung dieser Probleme als etwas schwieriger. Eine Implikation zwischen referenzierendem Feature und dem Feature, das das Ziel für die Referenz enthält, würde das referenzierende Feature als nicht mehr auswählbar kennzeichnen, wenn beide Abhängigkeiten (der gegenseitige Ausschluss und die Implikation) gelten sollen.

In dieser Arbeit wird ein familienbasierter Typchecker für feature-orientierte Softwareproduktlinien in Java entwickelt, der anhand des *Abstract Syntax Tree*s (einer abstrakten Repräsentation der Programmstruktur) und des Feature-Modells die Produktlinie auf Typfehler hin untersucht und entsprechende Fehlermeldungen ausgibt. Der Typchecker basiert auf Fuji[1]. Fuji ist ein erweiterbarer Compiler für Softwareproduktlinien, der auf dem erweiterbaren Java-Compiler JastAddJ[2] aufbaut.

Der Typchecker wird anhand einer Auswahl von 12 Softwareproduktlinien unterschiedlicher Größe aus der Beispielsammlung von Fuji evaluiert.

Abschließend erfolgt ein Vergleich der Performanz dieses Typcheckers mit dem naiven, produktbasierten Ansatz der Typprüfung mittels Fuji.

[1] `http://fosd.de/fuji/`, besucht am 2013-04-09

[2] `http://jastadd.org/web/jastaddj/`, besucht am 2013-04-09

## 1.1 Aufbau der Arbeit

- In Kapitel 2 werden zunächst die für das Verständnis der Arbeit wichtigen Begriffe erklärt.
- In Kapitel 3 wird erläutert, welche speziellen Probleme bei der Softwareproduktlinien-Programmierung aufgrund der Bedingungen im Feature-Modell auftreten können.
- In Kapitel 4 wird der generelle Lösungsansatz der vorliegenden Arbeit vorgestellt.
- Im Kapitel 5 werden die Implementierung des Typcheckers, die Modifikationen bestehender Dateien, und eine benötigte Hilfsdatei vorgestellt. Dieser Typchecker gibt bei auftretenden Problemen detaillierte Fehlermeldungen aus.
- In Kapitel 6 wird der entwickelte Typchecker an verschiedenen Beispielproduktlinien getestet und die Ergebnisse ausgewertet. Außerdem wird die Zeitmessung des Typcheckers erläutert und mit dem naiven Ansatz verglichen.
- In Kapitel 7 werden offene Probleme und Verbesserungen dieses Ansatzes angesprochen.
- In Kapitel 8 wird auf verwandte wissenschaftliche Arbeiten verwiesen.
- In Kapitel 9 werden die Testfälle angesprochen, die als Regressionstests für die Entwicklung des Typcheckers dienen.
- Abschließend erfolgt in Kapitel 10 eine Zusammenfassung dieser Arbeit.

# 2 Grundlagen

In diesem Abschnitt werden die Grundlagen angesprochen, die für das Verständnis der Arbeit wichtig sind. Dazu werden Softwareproduktlinien erklärt, die feature-orientierte Programmierung erläutert, auf Typsysteme eingegangen und Fuji, JastAddJ sowie FeatureIDE vorgestellt, auf denen die Typprüfung aufbaut.

## 2.1 Softwareproduktlinie

Clements und Northrop definieren eine Softwareproduktlinie als eine Menge von Softwaresystemen, die auf einer gemeinsamen Menge von Features aufbauen. Die einzelnen Systeme (Produkte) dieser Produktlinie werden mithilfe dieser Features erzeugt [CN02].

### 2.1.1 Feature

Eine Übersicht der vielen unterschiedlichen Definitionen, die für den Begriff *Feature* in der Literatur zu finden sind, liefert [AK09]. Daraus ist die folgende zusammenfassende Definition entnommen.

Ein Feature umfasst eine Funktionalitätseinheit eines Softwaresystems. Es erfüllt damit eine vorgegebene Anforderung oder bietet die Möglichkeit, das System an subjektive Bedürfnisse und Vorlieben anzupassen. Die Auswahl eines Features liefert auch eine Möglichkeit, ein System zu konfigurieren [AK09].

### 2.1.2 Feature-Modell

Die Beschränkungen bei der Kombination von Features werden mittels Abhängigkeiten zwischen Features in einem *Feature-Modell* beschrieben. Aus der Menge aller laut Feature-Modell möglichen Feature-Kombinationen ergeben sich die Produkte der Softwareproduktlinie. Ein Feature-Modell besteht aus einer hierarchisierten Menge von Features. Die möglichen Beziehungen zwischen einem Elternfeature und seinen Kindfeatures sind in Tabelle 1 aufgelistet [Bat05].

| Beziehung | Erklärung |
|---|---|
| Und | *Alle* Kindfeatures müssen ausgewählt werden |
| Alternative | *Genau ein* Kindfeature muss ausgewählt werden |
| Oder | *Mindestens ein* Kindfeature muss ausgewählt werden |
| Obligatorisch | Das Feature *muss* ausgewählt werden |
| Optional | Das Feature *kann* ausgewählt werden |

Tabelle 1: Beziehungen zwischen Elternfeature und Kindfeatures. Das Elternfeature ist dabei als ausgewählt zu betrachten [Bat05].

Den in einem Feature-Modell aufgelisteten Features liegt entweder eine Implementierung zugrunde (*konkrete* Features) oder sie dienen nur zur besseren Ausgestaltung der Hierarchie (*abstrakte* Features) [Thü+11].

Für die Repräsentation eines Feature-Modells stehen mehrere Varianten zur Verfügung, die sich alle ineinander überführen lassen [Bat05]. Diese verschiedenen Repräsentationen werden im Folgenden erläutert.

**Feature-Liste** Generell ist es möglich, alle gültigen Produkte mithilfe einer Liste von Features, aus denen ein Produkt zusammengesetzt wird, aufzuzählen [Thü+12]. Dies kann in der simpelsten Form in einer Textdatei erfolgen, in der in jeder Zeile ein Produkt durch die Konkatenation der beteiligten Features gekennzeichnet wird. Diese Repräsentation wird allerdings schnell unübersichtlich und ist daher nur für kleine Produktlinien geeignet.

**Feature-Diagramm** In einem Feature-Diagramm wird das Feature-Modell graphisch als Baum repräsentiert. Ein Beispiel ist in Abbildung 3 auf Seite 15 zu sehen. In diesem Beispiel wird auch die graphische Darstellung der Beziehungen aus Tabelle 1 gezeigt[1]. Weitere Einschränkungen (*Constraints*) von Feature-Kombinationen, die nicht oder nur sehr umständlich durch die in Tabelle 1 aufgelisteten Beziehungen angegeben werden können, können als aussagenlogische Ausdrücke angehängt werden [Bat05].

**Grammatik** Die hierarchischen Beziehungen zwischen Features können alternativ zur graphischen Darstellung auch mithilfe einer Grammatik repräsentiert werden. Diese findet z.B. in den Feature-Modell-Dateien für das Feature-Modell-Spezifikationswerkzeug `guidsl`[2] Verwendung.

**Aussagenlogische Formel** Eine Grammatik (mit Constraints) ist eine kompakte Darstellung einer aussagenlogischen Formel. Die beiden Darstellungen können aufeinander abgebildet werden [Bat05]. Die aussagenlogische Formel bildet dabei eine Konjunktion von Formeln. Mithilfe eines SAT-Solvers kann getestet werden, ob eine bestimmte Auswahl an Features die Gesamtformel erfüllt und dadurch ein valides Produkt der Produktlinie darstellt.

### 2.1.3 Implementierungen

Für die Implementierung von Softwareproduktlinien stehen verschiedene Verfahren zur Verfügung [AK09]. Bestimmte Teile des Quellcodes können durch Annotationen (z.B. `#ifdef`-Anweisungen für den C-Präprozessor) als zu bestimmten Features zugehörig gekennzeichnet werden. Durch bedingtes Kompilieren aufgrund der Auswahl von Features können diese so gekennzeichneten Teile in das zu erstellende Produkt miteinbezogen oder ausgeschlossen werden. Weitere Verfahren sind Software-Frameworks [JF88], Komponenten [SGM02] sowie aspekt- [Kic+97], feature- [Pre97] und delta-orientierte Programmierung [Sch+10]. Bei all diesen Verfahren werden Features Implementierungseinheiten zugeordnet [Hol12].

[1] Die Und-Beziehung wird durch Striche vom Elternfeature zu den Kindfeatures ohne weitere Kennzeichnung zwischen den Strichen dargestellt.

[2] `http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/guidsl.html`, besucht am 2013-04-09

Da in dieser Arbeit ein Typchecker für feature-orientierte Produktlinien zu entwickeln ist, wird im nächsten Abschnitt nur auf die feature-orientierte Programmierung weiter eingegangen.

## 2.2 Featureorientierte Programmierung

In der feature-orientierten Programmierung (FOP) wird Funktionalität in Feature-Modulen gekapselt. Durch die Kombination dieser Module werden die einzelnen Produkte erzeugt [Ape+10b].

### 2.2.1 Feature-Modul

In der objekt-orientierten Programmierung werden in Feature-Modulen Klassen oder Interfaces eingeführt oder Klassen oder Interfaces durch andere Feature-Module verändert, indem Methoden hinzugefügt oder überschrieben und Felder hinzugefügt werden [Ape+10b].

Alle Einführungen und Verfeinerungen von Klassen oder Interfaces eines Feature-Moduls werden als *Kollaboration* bezeichnet. Eine Einführung oder Verfeinerung einer Klasse oder eines Interfaces wird *Rolle* genannt (siehe Abbildung 1 mit den Features Search, Directed, Weighted und Colored).

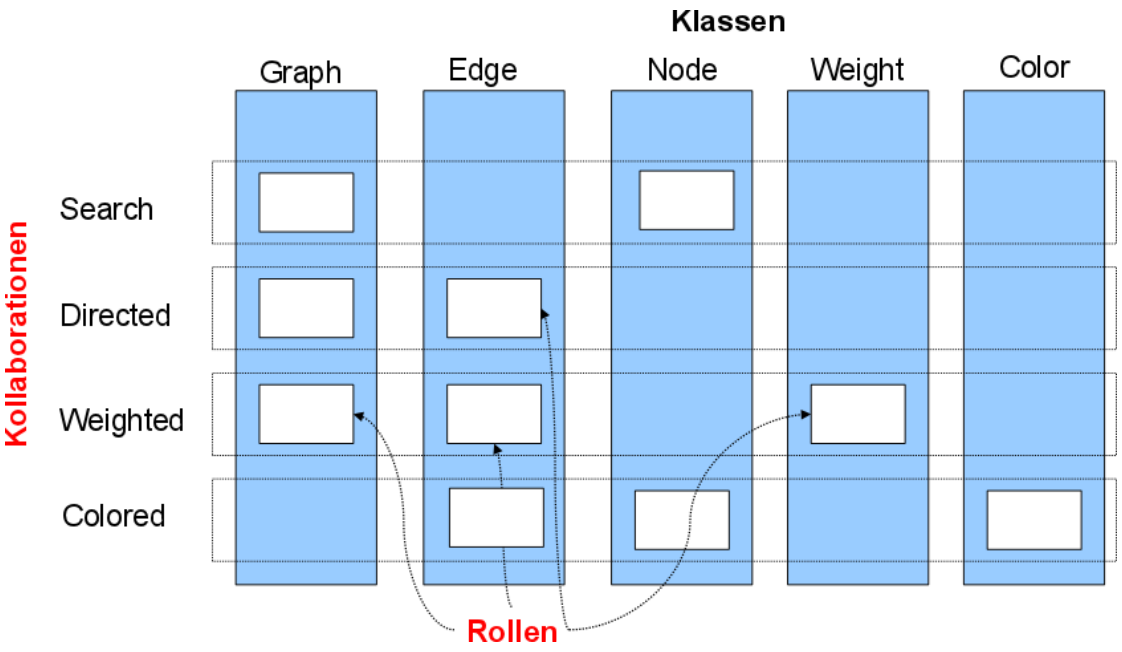



Abbildung 1: Kollaborationen und Rollen (entnommen aus [AKS11])

Abgesehen von dem Schlüsselwort `original()`, mit dem eine Version einer Methode vor deren Erweiterung aufgerufen wird, und einigen anderen möglichen Schlüsselwörtern

kann Standard-Java verwendet werden, um eine feature-orientierte Softwareproduktlinie in Java z.B. mit dem FOP-Werkzeug FeatureHouse[1] zu erstellen.

Aus einer Menge von Feature-Modulen wird der Quelltext des Produkts generiert. Dies wird als *Komposition* bezeichnet. Der entstandene Quelltext wird anschließend kompiliert (siehe Abbildung 2) [Hol12], [Kol11].

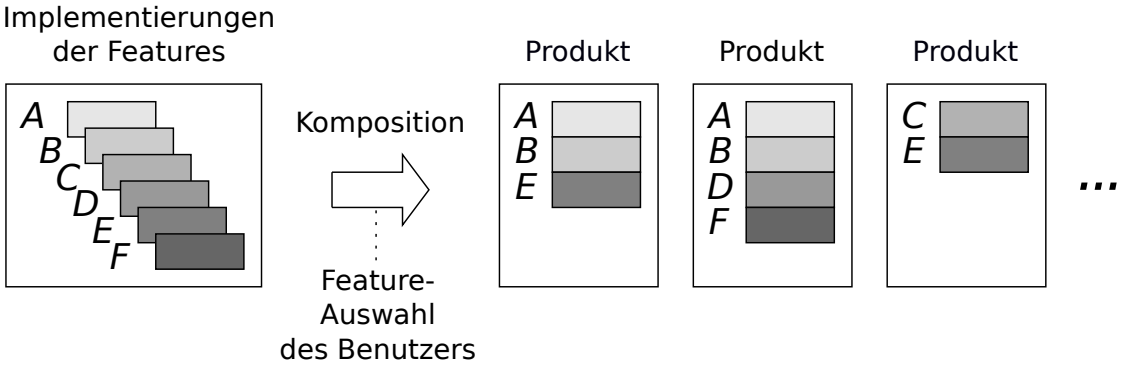



Abbildung 2: Feature-Komposition (entnommen aus [AK09])

## 2.3 Typsystem

Ein *Typsystem* ist eine leichtgewichtige formale Methode, um die Programmiererin dabei zu unterstützen, sicherzustellen, dass sich ein System in Bezug auf seine expliziten oder impliziten Spezifikationen korrekt verhält [Pie02]. Die syntaktischen Elemente des Programmtextes werden dabei anhand des Typs ihrer erzeugten Werte klassifiziert (in Typausdrücke zerlegt). Ein Typsystem besteht aus Inferenz-Regeln, aus denen sich alle solchen Ausdrücke ableiten lassen. Ein Typchecker überprüft, ob diese Ableitung durchgeführt werden kann. Ein Programm, das sich konform zu den Regeln des Typsystems verhält und bei der Typprüfung keine Fehler verursacht, wird als *wohlgetypt* bezeichnet. Dadurch lassen sich bestimmte fehlerhafte Verhaltensweisen des Programms ausschließen [Aho+07], [Pie02], [Hol12].

## 2.4 JastAddJ

JastAddJ, der *JastAdd Extensible Java Compiler*, ist ein erweiterbarer Java-Compiler, der mittels JastAdd implementiert wurde. JastAdd ist ein Meta-Kompilierungs-System, das *Reference Attribute Grammars* (RAGs, [Hed00]) unterstützt [EH07], [Jas]. JastAddJ lässt sich durch Aspekte erweitern, die Inter-Typ-Deklarationen beinhalten, die die Klassen und Interfaces des JastAddJ-*Abstract Syntax Tree*s statisch modular erweitern[2]. Auf diese Weise lässt sich nachträglich z.B. eine Typprüfung implementieren. Dazu werden vorhandene Klassen und Interfaces des *Abstract Syntax Tree*s in einem Aspekt erweitert,

[1] `http://www.fosd.de/fh`, besucht am 2013-04-15

[2] [Jas], `http://jastadd.org/web/documentation/concept-overview.php`, besucht am 2013-04-23

indem benötigte Felder oder Methoden zu diesen hinzugefügt werden. Diese Vorgehensweise ähnelt der aspekt-orientierten Programmierung, bei der querschneidende (über den Quelltext verstreute) Belange in Aspekten implementiert werden. Die Bestandteile dieses Aspektes werden bei der Kompilierung an den betreffenden Stellen eingewoben [Jas].

## 2.5 Fuji

Fuji ist ein Java-Compiler, der feature-orientierte Programmierung unterstützt. Der Quelltext wird dabei nicht wie bei anderen FOP-Werkzeugen wieder als Quelltext ausgegeben, sondern direkt in Java-Bytecode kompiliert. Dadurch werden bessere Fehler- und Typprüfungen ermöglicht. Durch die Erweiterbarkeit von Fuji können darauf aufbauend leicht neue Werkzeuge entwickelt werden. Fuji selbst baut auf JastAddJ auf [Kol11], [Fuj]. Der in dieser Arbeit entwickelte Typchecker ist eine Erweiterung von Fuji.

## 2.6 FeatureIDE

FeatureIDE ist ein Open-Source-Framework für die feature-orientierte Softwareentwicklung mit der Entwicklungsumgebung Eclipse [Thü+13].

Neben einigen anderen Implementierungsmethoden unterstützt FeatureIDE auch die feature-orientierte Programmierung [Thü+13]. Unter anderem lassen sich damit Feature-Modelle bearbeiten und anschaulich darstellen sowie Produkte einer Produktlinie erzeugen.

Aus dem Quelltext von FeatureIDE wird Funktionalität benötigt, die die Features, das Feature-Modell und die Abhängigkeiten zwischen Features behandelt. Mithilfe von Funktionen aus FeatureIDE werden aus dem Feature-Modell Beziehungen zwischen Features ermittelt, um sie mit den Abhängigkeiten, die sich aus der Implementierung ergeben, zu vergleichen.

# 3 Problemstellung

Eine Herausforderung bei der Erstellung von Produktlinien ist es, sicherzustellen, dass jedes valide Produkt (auf Grundlage einer laut Feature-Modell validen Auswahl von Features) ein typkorrektes Programm darstellt [Ape+10a]. Im Laufe der Entwicklung einer Produktlinie kann es dazu kommen, dass Abhängigkeiten zwischen den einzelnen Features, die durch die Implementierung entstehen, nicht mehr mit den Abhängigkeiten übereinstimmen, die das Feature-Modell vorgibt. Es kann dann laut Feature-Modell valide Produkte geben, die Typfehler enthalten. Eine Unterklasse von möglichen Typfehlern bilden die Referenzen, für die kein Ziel zur Verfügung steht (*dangling references*) [Ape+10a].

Aufgrund der enormen Anzahl an Produkten, die bei vielen unabhängigen Features in der Softwareproduktlinie entstehen, ist es gerade bei großen Produktlinien in der Regel nicht möglich, sämtliche Produkte zu komponieren und zu kompilieren. Manche Typfehler können zwar bei der Erstellung einiger Varianten erkannt werden, allerdings kann es passieren, dass ein Fehler unerkannt bleibt, da genau das Produkt nicht erzeugt wird, das diesen Fehler beinhaltet. Dadurch fallen diese Probleme möglicherweise erst sehr spät in der Entwicklung auf, wenn ein solches Produkt erzeugt wird. Dies verursacht in der Regel hohe Kosten [Thü+12].

## 3.1 Referenzen ohne Ziel

Wenn in der Implementierung eines Features $A$ eine Referenz auf ein Feld, eine Methode oder einen Typ existiert, das/die zwar in einem anderen Feature $B$ vorhanden ist, dieses Feature $B$ aber nicht in jedem validen Produkt enthalten ist, in dem Feature $A$ enthalten ist, dann kann es valide Produkte geben, bei denen die Referenz auf das benötigte Feld, die benötigte Methode oder den benötigten Typ ins Leere zeigt, da in diesem Produkt kein Ziel für diese Referenz vorhanden ist. Diese Situation kann auftreten, wenn alle Features, die ein Ziel für die Referenz enthalten, laut Feature-Modell optional sind oder alle in gegenseitigem Ausschluss zu dem Feature stehen, von dem aus die Referenz erfolgt. Bei Auftreten dieser Situation findet sich im Feature-Modell keine Bedingung, die bei Auswahl des referenzierenden Features die Auswahl eines Features, das ein Ziel für die Referenz enthält, erzwingen würde.

Es kann also ein Produkt geben, in dem kein einziges dieser Ziele zu erreichen ist. Ein Beispiel für so einen Typfehler wird im Folgenden erläutert.

In der Softwareproduktlinie **EPL** (eine feature-orientierte Java-Implementierung des *Expression Problem*, [LHB04]) wird im Feature Plus_ToString in der Klasse `Plus` auf die Variablen `x` und `y` zugegriffen (Zeile 5 des linken Listings), die nur im Feature Plus in der Klasse `Plus` eingeführt werden (Zeilen 4 und 5 im rechten Listing):

Feature PLUS_TOSTRING (optional):

```
1  package tmp;
2
3  public class Plus {
4    public String toString(){
5      return x + " + " + y;
6    }
7  }
```

Feature PLUS (optional):

```
1  package tmp;
2
3  public class Plus implements Exp {
4    Exp x;
5    Exp y;
6    Plus(  Exp x,  Exp y){
7      this.x=x;
8      this.y=y;
9    }
10 }
```

Sowohl Feature PLUS_TOSTRING als auch Feature PLUS sind im Feature-Modell als optional gekennzeichnet (Abbildung 3).

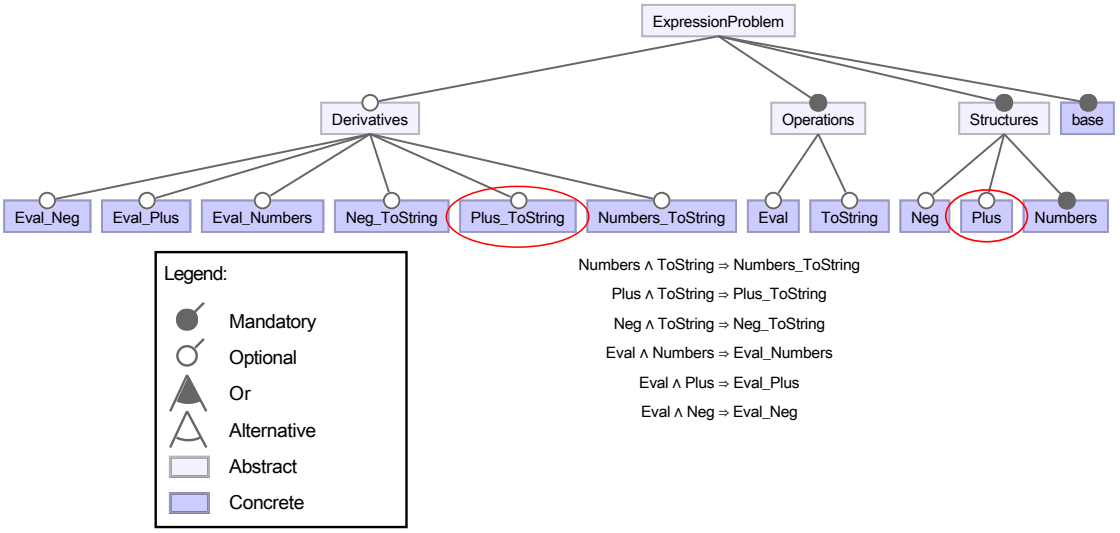



Abbildung 3: EPL-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13]). Die Features PLUS_TOSTRING und PLUS sind hervorgehoben. In der Legende (unten links) werden alle möglichen Beziehungen aufgeführt.

Wird nun eine laut Feature-Modell valide Variante erzeugt, in der das Feature PLUS_TOSTRING ausgewählt wird, das benötigte Feature PLUS jedoch nicht, dann fehlen diese Variablen in der aus den Features zusammengefügten Klasse `Plus` und die Zugriffe auf diese Variablen gehen ins Leere.

Im Feature-Modell findet sich kein Constraint, das erzwingen würde, dass bei einer Auswahl des Features PLUS_TOSTRING auch das Feature PLUS ausgewählt werden muss. Wird ein entsprechendes Constraint PLUS_TOSTRING ⇒ PLUS zum Feature-Modell hinzugefügt, so treten die beschriebenen Probleme nicht mehr auf.

Der Erstellerin dieser Softwareproduktlinie sollten diese Probleme mittels entsprechender Fehlermeldungen mitgeteilt werden:

```
.../EPL/features/Plus_ToString/tmp/Plus.java:5,12:
  Semantic Error: MAYBE dependency:
Feature Plus_ToString accesses the field
  Exp x;
of feature Plus.
Feature Plus may not be present in every valid selection.

.../EPL/features/Plus_ToString/tmp/Plus.java:5,24:
  Semantic Error: MAYBE dependency:
Feature Plus_ToString accesses the field
  Exp y;
of feature Plus.
Feature Plus may not be present in every valid selection.
```

Es wäre möglich, dass alle *Derivatives* dieser Softwareproduktlinie (und somit auch das *Derivative* Plus_ToString) in einer Benutzeroberfläche bei der Zusammenstellung eines Produkts als nicht auswählbar gekennzeichnet oder nicht angezeigt werden. Diese Einschränkung beruht dann aber auf einer Konvention, die der entwickelte Typchecker ohne weitere Formalisierung dieser Beschränkung nicht berücksichtigen kann. Eine deutlich „sauberere“ Lösung wäre an dieser Stelle die Modellierung mittels Äquivalenzen zwischen den *Derivatives* und den beteiligten *Operations* und *Structures*[1].

Diese Softwareproduktlinie enthält noch einige andere Typfehler, die durch Diskrepanzen zwischen der Implementierung und dem Feature-Modell hervorgerufen werden (siehe dazu den Abschnitt 6.1 auf Seite 35).

## 3.2 Fehlende Subtypinformationen

Bei der Analyse der Fehlerausgabe der Kompilierung der einzelnen Produkte der Softwareproduktlinie EPL zeigten sich auch Fehler, die auf eine andere Weise zustande kommen. Bei der Auswahl[2] der Features base, Numbers, Eval, Eval_Numbers und Eval_Neg wird bei der Kompilierung mit Fuji folgende Fehlermeldung ausgegeben:

```
/home/lutz/Benchmark/EPL/features/Eval/tmp/Test.java:7:
  Semantic Error: can not assign e of type tmp.Exp a value of type tmp.Neg
```

In der aus den Features base und Eval zusammenkomponierten Klasse `Test` wird in Zeile 6 des linken Listings versucht, der Variablen `e` vom Typ `Exp` eine Instanz der Klasse `Neg` zuzuweisen. Die Information, dass die Klasse `Neg` das Interface `Exp` implementiert, findet sich im nicht ausgewählten Feature Neg, das im Feature-Modell als optional gekennzeichnet ist, nicht jedoch im ausgewählten Feature Eval_Neg. Die Auswahl des Features Neg wird auch durch kein Constraint erzwungen. Dadurch steht diese Subtypinformation in diesem Produkt nicht zur Verfügung und der Kompiler kann die Zuweisung nicht durchführen[3].

[1] Ein ähnliches Problem tritt auch in der Softwareproduktlinie GPL auf (siehe Abschnitt 6.2 auf Seite 38). Dort wurden diese *Derivatives* im Feature-Modell als *hidden* gekennzeichnet.

[2] Erzeugtes Produkt `EPL/products/Variant0030` im Benchmark-Ordner der beiliegenden CD

[3] Die Instanziierung der Klasse `Neg` schlägt in diesem Produkt auch fehl, da der passende Konstruktor nicht zur Verfügung steht. Dieses Problem ensteht aber durch ein Ziel in einem optionalen Feature. Diese Art von Typfehlern wurde bereits im vorhergehenden Abschnitt 3.1 besprochen.

Feature Eval (optional)[a]:

```
1 package tmp;
2
3 public class Test {
4   static void evaltest(){
5     /* ... */
6     [Exp] e=new Neg(new Num(1));
7     /* ... */
8   }
9 }
```

[a] „[Exp]“ aus Feature BASE

Feature Neg (optional):

```
1 package tmp;
2
3 public class Neg implements Exp {
4   /* ... */
5 }
```

Feature Eval_Neg (optional):

```
1 package tmp;
2
3 public class Neg {
4   /* ... */
5 }
```

Aus diesem Beispiel lässt sich ableiten, dass neben den Problemen, die in Abschnitt 3.1 beschrieben werden, weitere Probleme auftreten können. Zwar ist in diesem Fall wenigstens ein Feature mit referenziertem Ziel in allen Produkten, die diese Referenz enthalten, vorhanden; implizite oder explizite Casts schlagen in diesen Produkten aber fehl, da eine benötigte Subtypinformation nur in Features vorhanden ist, die alle optional sind oder alle im gegenseitigen Ausschluss zu dem Feature stehen, das diese Subtypinformation benötigt. Außerdem findet sich kein Constraint im Feature-Modell, das bei Auswahl des Features, das die Subtypinformation benötigt, eine Auswahl eines Features mit der benötigten Subtypinformation erzwingen würde. Es kann also der Fall auftreten, dass die benötigte Subtypinformation in einem laut Feature-Modell validen Produkt gar nicht zur Verfügung steht.

Wird im Beispiel ein passendes Constraint Eval_Neg ⇒ Neg zum Feature-Modell hinzugefügt, so tritt das beschriebene Problem nicht mehr auf.

Der Erstellerin der Softwareproduktlinie sollte ein derartiges Problem analog zu den Problemen bei fehlenden Zielen für Referenzen mittels einer entsprechenden Fehlermeldung mitgeteilt werden:

```
.../EPL/features/Eval/tmp/Test.java:7:
  Semantic Error: MAYBE dependency:
Class Neg of feature Eval_Neg accesses (by an implicit cast/in the type hierarchy) the type
  public interface Exp { ... }
which is accessible via interface Exp.
Class Neg of feature Neg implements interface Exp.
The information that class Neg implements interface Exp is only present in feature Neg.
Feature Neg may not be present in every valid selection.
```

# 4 Genereller Lösungsansatz

Um die im vorangehenden Abschnitt angesprochenen Probleme anzugehen, werden für alle betrachteten Zugriffe (für die genaue Auflistung siehe unten) das Quellfeature (das Feature, aus dem heraus der Zugriff stattfindet) und die Zielfeatures (die Features, die Ziele für diese Zugriffe enthalten) betrachtet und mithilfe der aus dem Feature-Modell ermittelten Abhängigkeiten festgestellt, ob bei einer Auswahl des Quellfeatures immer auch mindestens ein Zielfeature ausgewählt werden muss. Ist dies nicht der Fall (weil die Zielfeatures alle optional sind oder alle mit dem Quellfeature in gegenseitigem Ausschluss stehen), dann wird eine entsprechende Fehlermeldung ausgegeben. Damit werden die in Abschnitt 3.1 angesprochenen Probleme adressiert.

Daneben werden auch die Probleme betrachtet, die durch fehlende Subtypinformationen verursacht werden (siehe Abschnitt 3.2).

Möglicherweise steht zwar mindestens ein Zielfeature in jedem Produkt, in dem dessen Ziel referenziert wird, zur Verfügung. Die Referenz ist allerdings nur aufgrund einer Subtypbeziehung der beteiligten Typen möglich. Die dazu benötigte Subtypinformation ist nur in Features vorhanden, die bei Auswahl des Features, das die Subtypinformation benötigt, alle optional sind oder in gegenseitigem Ausschluss zu diesem stehen.

Für derartige Typfehler werden ebenfalls entsprechende Fehlermeldungen ausgegeben.

Um die angesprochenen Probleme ermitteln zu können, ist es notwendig, die Beziehungen der Features zu kennen, die sich aus dem Feature-Modell ergeben. Dazu wird aus den Abhängigkeiten des Feature-Modells mittels eines SAT-Solvers ermittelt, welche logischen Schlüsse sich aus der Auswahl eines Features auf das Vorhandensein der anderen Features in validen Produkten ergeben. Dabei gibt es die folgenden Beziehungen (basierend auf FeatureIDE [Thü+13]):

- Ein Feature $A$ steht mit einem anderen Feature $B$ in *always*-Beziehung, wenn sich aus dem Vorhandensein des Features $A$ laut Feature-Modell logisch zwingend das Vorhandensein des Features $B$ folgern lässt.[1]
- Ein Feature $A$ steht mit einem anderen Feature $B$ in *maybe*-Beziehung, wenn sich aus dem Vorhandensein des Features $A$ laut Feature-Modell logisch ergibt, dass das Feature $B$ vorhanden sein kann, aber nicht zwingend vorhanden sein muss.
- Ein Feature $A$ steht mit einem anderen Feature $B$ in *never*-Beziehung, wenn sich aus dem Vorhandesein des Features $A$ laut Feature-Modell logisch folgern lässt, dass das Feature $B$ nicht ausgewählt werden kann.

Diese Beziehungen zwischen Features spielen eine zentrale Rolle bei der Ermittlung von Referenzen, die zu Typfehlern in einigen Produkten führen. So erzeugt eine Referenz eines Features auf ein Feld in einem anderen Feature, das mit dem ersten in *maybe*-Beziehung steht, einen Typfehler in validen Produkten, in denen das Quellfeature, nicht aber das Zielfeature für diese Referenz ausgewählt wird.

[1] Diese Beziehung wird bei der Implementierung des Typcheckers im Rahmen dieser Arbeit nicht benötigt.

In dieser Arbeit werden nur Fehler behandelt, die sich aus Diskrepanzen zwischen der Implementierung und dem Feature-Modell ergeben, nicht aber syntaktische oder semantische Typfehler, die auch ohne Produktlinien-Ansatz auftreten würden. Fehlt z.B. ein Ziel für einen Zugriff in der gesamten Quelltextbasis, so beruht dies nicht darauf, das ein Feature mit einem entsprechenden Ziel in einem konkreten Produkt möglicherweise fehlen würde. Demnach erfolgt auch kein Vorschlag zur Änderung von eventuell falsch geschriebenen Klassennamen, Methodennamen, etc. Dies wird von [Hol12] zur Korrektur von Typfehlern vorgeschlagen.

Bei mehreren zur Verfügung stehenden Zielen erfolgt keine Prüfung, ob sich diese Ziele alle ausschließen, d.h. ob in einem erzeugten Produkt nur jeweils ein Ziel zur Verfügung steht und somit keine doppelten Felder oder Methoden in diesselbe Klasse kompiliert werden. Dazu müsste mittels des Feature-Modells sichergestellt werden, dass nicht nur mindestens ein Ziel für eine Referenz in allen validen Produkten, in denen dieses Ziel benötigt wird, zur Verfügung steht, sondern auch, dass maximal ein solches Ziel zur Verfügung steht. Diese Überprüfung liegt außerhalb des Rahmens der vorliegenden Arbeit.

Der Typchecker, der in dieser Arbeit entwickelt wird, ist als eine Erweiterung von Fuji anzusehen. Der Typchecker benötigt die Erweiterung von Fuji, die die Referenzen im *Abstract Syntax Tree* ermittelt, d.h. welche Felder, Methoden oder Typen von welchen Features benutzt (referenziert) werden. Diese Erweiterung wird mit der Option `-fopRefs` aktiviert und analysiert die gesamte Produktlinie mit allen Features. Für diese Analyse wird eine andere Kompositionsstrategie verwendet als für die Komposition eines validen Produkts, da z.B. aus alternativen Features verschiedene Implementierungen für Methoden mit gleicher Signatur eingeführt werden, was laut den Typregeln von Java nicht erlaubt ist und Fehlermeldungen produzieren würde. Für die Analyse der Referenzen müssen aber alle Features zusammenkomponiert werden [Kol11].

Der Typchecker überprüft Zugriffe auf

- Felder, auch mit dem Vorsatz `super` (expliziter Zugriff auf ein Feld der Oberklasse),
- Methoden (inklusive des Zugriffs mittels `original()` auf die Version einer Methode vor deren Erweiterung),
- Konstruktoren, sowie den Zugriff auf die Konstruktoren der Oberklasse, und
- Typen.

Bei all diesen Zugriffen wird auch überprüft, ob diese Zugriffe eventuell gar nicht möglich sind, da benötigte Subtypinformationen nicht in jedem validen Produkt, das diesen Zugriff enthält, vorhanden sind.

Bei folgenden Typumwandlungen erfolgt außerdem eine Überprüfung, ob eine benötigte Subtypinformation nicht in jedem validen Produkt, das diese Subtypinformation benötigt, zur Verfügung steht:

- explizite und
- implizite Casts
  - bei einer Zuweisung mit unterschiedlichen Typen,
  - bei unterschiedlichen Typen zwischen einem übergebenen Argument und dem erwarteten Parameter einer Methode (auch für den speziellen Methodenaufruf `original()` und für einen Konstruktor) und
  - bei unterschiedlichen Typen des übergebenen und erwarteten Rückgabewerts einer Methode

# 5 Implementierung

## 5.1 Beschreibung des Aspekts `ExtendedTypeCheck.jrag`

Die eigentliche Typprüfung wird im Aspekt `ExtendedTypeCheck.jrag` implementiert. Dieser Aspekt beinhaltet Inter-Typ-Deklarationen, die die Klassen und Interfaces des JastAddJ-*Abstract Syntax Tree*s statisch modular erweitern[1].

Um eine neue Methode in die Klasse `ASTNode`, dem obersten Typ des *Abstract Syntax Tree*s, einzufügen, ist es ausreichend, folgende Methode in den Aspekt zu schreiben. Diese Methode wird automatisch in die entsprechende Klasse eingewoben. Zu beachten ist die spezielle Signatur dieser Methode, die die Zugehörigkeit zu `ASTNode` kennzeichnet.

```
1  public void ASTNode.collectSplErrors() {
2    splTypeCheck();
3    for(int i = 0; i < getNumChild(); i++) {
4      getChild(i).collectSplErrors();
5    }
6  }
```

Für die im Abschnitt 4 „Genereller Lösungsansatz“ auf Seite 18 aufgezählten Zugriffe wird im *Abstract Syntax Tree* rekursiv von oben nach unten gehend für die einzelnen Knoten mittels der Methode `splTypeCheck()` die Typüberprüfung initialisiert. Dabei werden wichtige Informationen über diese Zugriffe gesammelt und für die weitere Analyse an die Methode `addSplErrors` in der Klasse `ASTNode` übergeben.

Die betreffenden Knoten sind im Abschnitt 5.1.4 auf Seite 29 aufgelistet.

Die zu sammelnden Informationen sind im Abschnitt 5.1.3 auf Seite 24 aufgelistet.

In der Methode `addSplErrors` wird der Zugriff daraufhin überprüft, ob die beteiligten Features in einer Beziehung zueinander stehen, die dazu führen könnte, dass in einem validen Produkt für den Zugriff benötigte Ziele fehlen, da die Features, die diese Ziele enthalten, nicht hinzukomponiert wurden. Ist dies der Fall, so wird eine entsprechende Fehlermeldung ausgegeben.

In einem zweiten Schritt wird analysiert, ob für diesen Zugriff Subtypinformationen benötigt werden. Sind diese Subtypinformationen nur in Features vorhanden, die nicht in jedem validen Produkt, in dem die Subtypinformation benötigt wird, enthalten sind, so wird für diesen Fall ebenfalls eine Fehlermeldung ausgegeben.

Da sich diese Subtypbeziehung des Typs, der die Subtypinformation benötigt, und des obersten referenzierten Typs über mehrere Ebenen der Typhierarchie erstrecken kann, muss die Typhierarchie von unten nach oben durchlaufen werden. In diesem Fall handelt es sich also nicht um eine direkte Subtypbeziehung, sondern um eine indirekte[2]. Bei jedem Übergang von einer Ebene zur nächsthöheren Ebene der Typhierarchie könnten die genannten Probleme auftauchen und werden deshalb einzeln geprüft und gegebenfalls gemeldet.

Die Pfade von unten nach oben können sich dabei aufgrund alternativer Subtypinformationen in verschiedenen Features verzweigen. So könnte beispielsweise eine Klasse

[1] [Jas], `http://jastadd.org/web/documentation/concept-overview.php`, besucht am 2013-04-23

[2] Typ $A$ erweitert indirekt den Typ $C$, wenn Typ $A$ den Typ $B$ erweitert und Typ $B$ den Typ $C$ erweitert (oder bei einer noch längeren Kette von Erweiterungen).

in einem Feature eine andere Klasse erweitern als in einem anderen (dazu alternativen) Feature. Letztendlich muss in jedem validen Produkt, in dem die Subtypinformation für den Zugriff benötigt wird, die Information über wenigstens einen kompletten Pfad in der Typhierarchie bereitstehen.

Zugriffe auf *Typen* werden beim Durchlauf des *Abstract Syntax Trees* zunächst nur gesammelt, und erst anschließend nach dem Durchlauf des *Abstract Syntax Trees* ausgewertet. Dadurch können mehrmalige Referenzen in derselben Klasse desselben Features auf einen Typ gesammelt und diese Referenzen gemeinsam ausgewertet werden. Damit werden viele gleichlautende Fehlermeldungen vermieden, die sich nur in den Zeilen- und Spaltennummern unterscheiden würden. Außerdem findet dadurch nur jeweils eine Anfrage an den SAT-Solver (in der Methode `addSplErrors`) statt, wodurch die Laufzeit des Typcheckers verringert wird. Diese Vorgehensweise ist sinnvoll, da die Typen in einer Klasse eines Features in der Regel mehr als einmal referenziert werden (im Gegensatz zu Feldern und Methoden, die in der Regel weniger oft referenziert werden)[1].

Es wurde darauf Wert gelegt, dass die Fehlermeldungen möglichst aussagekräftig sind, um der Erstellerin der Softwareproduktlinie möglichst viele Informationen zu geben, so dass die Probleme rasch gefunden und behoben werden können. Allerdings ist die Ausgabe der Fehlermeldungen überwiegend generisch gehalten, um auch die Überprüfung auf benötigte Subtypinformationen in einem zweiten Schritt durch nochmaliges Durchlaufen der Hauptmethode `addSplErrors` analog zur Überprüfung auf referenzierte Ziele durchführen zu können.

### 5.1.1 Modifizierte Methoden der allgemeinen Fehlerüberprüfung

Die Methoden in diesem Abschnitt wurden aus der allgemeinen Fehlerüberprüfung von JastAddJ (aus dem Aspekt `ErrorCheck.jrag` in der Bibliothek `JastAddJ/Java1.4-Frontend`) herauskopiert und modifiziert, indem jeweils ein „`spl`“ oder „`Spl`“ vor Methodennamen, in Methodennamen oder in Strings ergänzt wurde. Außerdem wird das Feature-Modell in der Klasse `ASTNode` gespeichert. Dadurch wird analog zur üblichen Fehlerüberprüfung von JastAddJ nach der Komposition eines Produkts nun durch den Aufruf der Methode `splTypeCheck()` in den einzelnen Knoten des *Abstract Syntax Trees* eine Typprüfung rekursiv von oben nach unten durchgeführt.

**Starten der Typprüfung** Der Methode `splErrorCheck` der Klasse `Program` wird das Feature-Modell übergeben, das in der Klasse `ASTNode` gespeichert wird. Diese Methode wird in der Klasse `Main.java` von Fuji aufgerufen und startet somit die Typprüfung.

Der Quelltext der Softwareproduktlinie wird beim Parsen in `CompilationUnit`s unterteilt. Diese `CompilationUnit`s werden in der Klasse `Program` gespeichert.

Die Liste der `CompilationUnit`s dieses `Program`s wird durchgegangen, wobei die auftretenden Fehlermeldungen und Warnungen der `CompilationUnit` gesammelt werden.

[1] Erfahrungswerte des Autors dieser Masterarbeit. Diese Aussage ist diskutabel. In der Softwareproduktlinie TankWar erfolgte z.B. ein oftmaliger Aufruf derselben Methode kurz hintereinander in einer Fallunterscheidung (siehe die komplette Liste der Fehlermeldungen für TankWar im Anhang auf der beiliegenden CD). Prinzipiell wäre die Vorgehensweise für Typen auch für Felder und Methoden möglich.

Anschließend werden die bei der Überprüfung der `CompilationUnit`s gesammelten Typzugriffe in einem getrennten Schritt mit dem Aufruf der Methode `collectSplTypeErrors` ausgewertet (in Zeile 12 des folgenden Listings; siehe Abschnitt 5.1.4 „Einleiten der Überprüfung der Typzugriffe“ auf Seite 31; `Program` erweitert `ASTNode`).

```
1  public void Program.splErrorCheck(FeatureModel model, Collection collection,
2      Collection warn) {
3    setFeatureModel(model);
4    for(Iterator iter = compilationUnitIterator(); iter.hasNext(); ) {
5      CompilationUnit cu = (CompilationUnit)iter.next();
6      if(cu.fromSource()) {
7        cu.collectSplErrors();
8        collection.addAll(cu.errors);
9        warn.addAll(cu.warnings);
10     }
11   }
12   collectSplTypeErrors();
13   collection.addAll(errors);
14   warn.addAll(warnings);
15 }
```

**Rekursiver Aufruf der Typprüfung im *Abstract Syntax Tree*** Während bei der allgemeinen Fehlerüberprüfung von JastAddJ diverse Prüfmethoden in den einzelnen Knoten des *Abstract Syntax Tree* aufgerufen werden, findet hier nur noch der Aufruf der Methode `splTypeCheck()` statt.

Die Methode `collectSplErrors()` in der Klasse `ASTNode` geht den *Abstract Syntax Tree* von oben nach unten rekursiv durch und ruft auf den einzelnen Knoten jeweils die Methode `splTypeCheck()` auf, die für die Zugriffe Überprüfungen durchführt (siehe den Abschnitt 5.1.4 „Überprüfung der einzelnen Zugriffe“ auf Seite 29 für die speziellen Methoden).

Diese Methode wird in der oben beschriebenen `splErrorCheck`-Methode der Klasse `Program` aufgerufen (Die Klasse `CompilationUnit` erweitert die Klasse `ASTNode`).

```
1 public void ASTNode.collectSplErrors() {
2   splTypeCheck();
3   for(int i = 0; i < getNumChild(); i++) {
4     getChild(i).collectSplErrors();
5   }
6 }
```

### 5.1.2 Hilfsstrukturen

Fehlermeldungen, die Typzugriffe betreffen, sollen mehrere Dateinamen mit Zeilen- und Spaltennummern für all die Stellen enthalten, an denen ein Zugriff auf diesen Typ erfolgt. Diese Stellen des Aufrufs liegen dabei alle innerhalb desselben Typs. Um diese Fehlermeldungen ausgeben zu können, werden zwei modifizierte Methoden des Aspekts `ErrorCheck.jrag` benötigt:

- `ASTNode.typeError(String accessSourceInfo, String msg)` und
- `Problem.Problem(String fileName, String message, Severity severity, Kind kind)`.

Der Dateiname und die Zeilen- und Spaltennummer werden nun nicht mehr wie bei den bereits im Aspekt `ErrorCheck.jrag` vorliegenden Methoden beim Aufruf der Methode ermittelt, sondern explizit im String `accessSourceInfo` bzw. `fileName` zusätzlich zur eigentlichen Fehlermeldung übergeben. Dadurch können mehrere Zugriffsstellen mit der eigentlichen Fehlermeldung ausgegeben werden.

### 5.1.3 Überprüfung der Zugriffe

**Speichern des Feature-Modells** Das Feature-Modell wird in der Klasse `ASTNode` gespeichert.

**Konstanten** Um die Subtyp-Beziehung eines Typs, von dem aus der Zugriff stattfindet, zu einem Typ, der das Ziel für diesen Zugriff enthält, zu unterscheiden, werden vier Konstanten eingeführt, um eine angepasste Ausgabe der Fehlermeldung zu ermöglichen. Dabei wird unterschieden zwischen

- keiner Beziehung (da die Überprüfung auf möglicherweise fehlende Subtypinformationen in dieser Phase der Überprüfung noch nicht stattfindet),
- Unterklasse,
- Implementierung eines Interfaces und
- Sub-Interface.

***always*-Beziehung** Mittels der `FeatureDependencies`, die aus dem Feature-Modell ermittelt werden, wird in der Methode `ASTNode.always(FeatureDependencies dep, Feature fromFeature, Feature toFeature)` analysiert, ob ein Quellfeature `fromFeature` in *always*-Beziehung zu einem Zielfeature `toFeature` steht.

***never*-Beziehung** Mittels der `FeatureDependencies`, die aus dem Feature-Modell ermittelt werden, wird in der Methode `ASTNode.never(FeatureDependencies dep, Feature fromFeature, Feature toFeature)` analysiert, ob ein Quellfeature `fromFeature` in *never*-Beziehung zu einem Zielfeature `toFeature` steht.

***maybe*-Beziehung** Mittels der `FeatureDependencies`, die aus dem Feature-Modell ermittelt werden, wird in der Methode `ASTNode.maybe(FeatureDependencies dep, Feature fromFeature, Feature toFeature)` analysiert, ob ein Quellfeature `fromFeature` in *maybe*-Beziehung zu einem Zielfeature `toFeature` steht.

**Überprüfung der Zugriffe** Die Hauptmethode `ASTNode.addSplErrors` der Typprüfung nimmt folgende Argumente:

`int fromFeatureID` Die ID des Features, aus dem heraus der Zugriff durchgeführt wird.

`HashSet<Integer> contextPresentFeatures` Eine Liste der Features (anhand ihrer ID), die als ausgewählt betrachtet werden (das zugreifende Feature ist darin trivialerweise enthalten). Diese Liste ist wichtig für die Überprüfung auf fehlende

Subtypinformationen, um im Feature-Modell alle Features ermitteln zu können, deren Auswahl durch diese Liste an Features mittels Constraints erzwungen wird.

**`HashMap<ASTNode, TypeDecl> declsAndTargetHostTypes`** Die Ziele für den Zugriff zusammen mit den die Ziele umgebenden Typen in einer `HashMap` (Die Speicherung der die Ziele umgebenden Typen ist an dieser Stelle nötig, da es danach nur schwer möglich ist, aus den Zielen die sie umgebenden Typen zu ermitteln. Die Ziele werden als `ASTNode` gespeichert, um generisch Ziele mit unterschiedlichen Typen wie z.B. `BodyDecl` und `ClassInstanceExpr` behandeln zu können. Diese verschiedenen Typen besitzen aber keine gemeinsame Oberklasse mit einer `hostType()`-Methode, um den das Ziel umgebenden Typ zu ermitteln. Deshalb müsste später eine Fallunterscheidung hinsichtlich des Typs durchgeführt werden, um die umgebenden Typen mittels einer konkreten Methode dieses Typs zu ermitteln). Der umgebende Typ der Ziele wird für die Ausgabe benötigt.

**`String targetStructure`** Eine Beschreibung der Art des Zugriffs. Diese Beschreibung wird für die Ausgabe benötigt.

**`boolean isTypeAccess`** Ob es sich um einen Typzugriff handelt. Das ist wichtig für die Ausgabe, da für einen Typzugriff die zuvor in der Datei `TypeErrorInfo.java`[1] gespeicherten Dateinamen und Zeilen- und Spaltennummern der Stellen der Zugriffe ausgegeben werden.

**`TypeDecl sourceHostType`** Den Typ, aus dem heraus der Zugriff erfolgt. Dieser Typ wird für die Ausgabe benötigt.

**`boolean secondRoundDone`** Ob die zweite Runde (die Überprüfung auf möglicherweise fehlende Subtypinformationen) bereits abgeschlossen wurde. In diesem Fall muss die Auswertung der zweiten Runde nicht mehr stattfinden.

**`int state`** Die Art der Subtyp-Beziehung zwischen zwei zu überprüfenden Typen (siehe die Konstanten in Abschnitt 5.1.3). Dies ist wichtig für die Ausgabe bei der Überprüfung auf möglicherweise fehlende Subtypinformationen.

**`HashMap<ASTNode, TypeDecl> outputDeclsAndTargetHostTypes`** Informationen über die Ziele, zusammen mit den die Ziele umgebenden Typen in einer `HashMap`. Diese Informationen werden für die Ausgabe angepasst und unterscheiden sich daher von den Zielen im Parameter `declsAndTargetHostTypes`.

**`boolean isUnqualifiedAccess`** Ob es sich um einen Zugriff ohne qualifizierenden Vorsatz handelt, der angibt, wo das Ziel dieses Zugriffs zu finden ist.

**`TypeDecl narrowType`** Den genauesten Typ, der bei der Deklaration einer Variablen oder eines Feldes angegeben wird, wenn auf ein Feld, eine Methode oder einen Typ dieser Variable oder dieses Feldes zugegriffen wird. Dies ist wichtig für die Ermittlung der Verfügbarkeit der Informationen über die Subtypbeziehung des Typs dieser Variablen oder dieses Feldes.

[1] siehe den Abschnitt 5.3 auf Seite 33

`TypeDecl accessingType` Den Typ, aus dem heraus der Zugriff erfolgt. Dies ist wichtig für die Ausgabe und wird an dieser Stelle nochmal übergeben, da diese Information im Parameter `sourceHostType` im Laufe des zweiten Schritts überschrieben wird.

`HashMap<ASTNode, TypeDecl> accessingDeclsAndTargetHostTypes` Die Ziele in `declsAndTargetHostTypes` für den Zugriff zusammen mit den die Ziele umgebenden Typen in einer `HashMap`. Bei der Überprüfung auf fehlende Subtypinformation werden die Ziele im Parameter `declsAndTargetHostTypes` bei der Analyse im zweiten Schritt durch die Ziele eines Typzugriffs ersetzt. Dieser Typzugriff erfolgt von einem Typ, der die Subtypinformation benötigt, auf einen Typ, der diese Subtypinformation hat. Die Informationen über den eigentlich zu analysierenden Zugriff, die für die Ausgabe benötigt werden, werden deshalb hier nochmal gespeichert.

Aus dem Feature-Modell werden die konkreten Features der Softwareproduktlinie und deren Abhängigkeiten ermittelt. Mithilfe dieser Abhängigkeiten wird analysiert, ob für die Zugriffe in jedem validen Produkt, in dem dieser Zugriff vorhanden ist, mindestens ein Ziel in einem Feature dieses Produkts verfügbar ist. Dies kann dadurch geschehen, dass dieses Zielfeature im Feature-Modell als obligatorisch gekennzeichnet ist oder durch Constraints bei Auswahl des Quellfeatures logisch zwingend ausgewählt werden muss.

Einige Zugriffe können nur aufgrund einer Subtypbeziehung zweier beteiligter Typen durchgeführt werden. In einem zweiten Schritt wird deshalb überprüft, ob diese Information über die Subtypbeziehung in mindestens einem Feature vorhanden ist, das in jedem validen Produkt enthalten ist, in dem ein Feature diese Subtypinformation benötigt.

Bei einer indirekten Subtypbeziehung über mehrere Ebenen der Typhierarchie ist die Information in mehreren Typen in möglicherweise verschiedenen Features vorhanden. In diesem Fall muss die gesamte Typhierarchie zwischen den zwei betrachteten Typen durchgegangen werden, um sicherzustellen, dass die gesamte Subtypinformation in jedem Produkt, in dem sie benötigt wird, vorhanden ist.

Es wird eine Fallunterscheidung durchgeführt, wieviele Ziele für den Zugriff in der gesamten Produktlinie gefunden wurden[1]:

**Bei nur einem Ziel** wird bei unterschiedlichem Quell- und Zielfeature die Beziehung zwischen diesen Features ermittelt. Falls mehr als ein Feature als ausgewählt betrachtet wird[2], wird mittels der Methode `mayBeMissing` des Feature-Modells überprüft, ob das Zielfeature bei Auswahl der Features, die als ausgewählt betrachtet werden, optional ist oder in gegenseitigem Ausschluss zu einem der als ausgewählt betrachteten Features steht. Wird nur ein Feature als ausgewählt betrachtet, so erfolgt die Anfrage an das Feature-Modell mittels der Methoden `maybe` und `never`,

[1] Es wird nicht überprüft, ob gar kein Ziel vorhanden ist. Dies ist Aufgabe der allgemeinen Typüberprüfung, da dieser Typfehler nicht davon abhängt, welche Abhängigkeiten im Feature-Modell vorgegeben werden. Die Typprüfung der vorliegenden Arbeit konzentriert sich auf Typfehler, die durch Diskrepanzen zwischen dem Feature-Modell und der Implementierung hervorgerufen werden.

[2] Bei der Überprüfung auf fehlende Subtypinformationen werden das Feature, das den eigentlichen Zugriff enthält, und das Feature, das die Subtypinformation benötigt, als ausgewählt betrachtet

die ermitteln, ob bei Auswahl des Quellfeatures das Zielfeature optional ist bzw. in gegenseitigem Ausschluss zu diesem steht.

**Bei mehr als einem Ziel** wird mittels der Methode `mayBeMissing` des Feature-Modells ermittelt, ob bei Auswahl der Features, die als ausgewählt betrachtet werden, alle Zielfeatures optional sind oder in gegenseitigem Ausschluss zu einem Feature stehen, das als ausgewählt betrachtet wird.

In all diesen Fällen wird eine Fehlermeldung ausgegeben, die die Erstellerin der Produktlinie darüber informiert, an welcher Stelle der Implementierung ein Zugriff einen Typfehler erzeugt.

Für den zweiten Schritt (die Überprüfung, ob für den Zugriff benötigte Subtypinformationen vorhanden sind) wird, sofern dieser zweite Schritt noch nicht durchgeführt wurde, unterschieden, ob der Zugriff mit einem qualifizierenden Vorsatz durchgeführt wurde. Ist dies der Fall, so kann nicht davon ausgegangen werden, dass das Ziel des Zugriffs sich im gleichen Typ oder einem Obertyp des Typs, von dem aus der Zugriff erfolgt, befindet[1]. In dieser Situation wird ein neuer Typzugriff betrachtet, der aus dem genauesten Typ heraus erfolgt, der die Subtypinformation benötigt. In der Typhierarchie werden die Pfade ermittelt, über die dieser genaueste Typ mit seinem in Subtypbeziehung stehenden Obertyp verbunden ist. Der neue Typzugriff zielt auf alle Typen, die Obertypen dieses genauesten Typs sind und die in der Typhierarchie auf einem Pfad zu dem Typ liegen, der mit dem genauesten Typ in Subtypbeziehung steht. Mittels der Methode `splErrorCheckAccessToSuperType` wird rekursiv ausgehend vom Beginn jedes Pfades überprüft, ob die gesamte Subtypinformation in allen validen Produkten, in denen diese Subtypinformation benötigt wird, vorhanden ist.

**Rekursive Überprüfung in der Typhierarchie** Die Methode `ASTNode.splErrorCheckAccessToSuperType` führt die rekursive Überprüfung für den zweiten Schritt der Hauptmethode `ASTNode.addSplErrors` durch.

Abhängig davon, ob eine Klasse eine Klasse erweitert, eine Klasse ein Interface implementiert oder ein Interface ein Interface erweitert, werden alle Features ermittelt, die zu dem in der Typhierarchie unten stehenden Typen beitragen, und alle Typen der nächsthöheren Ebene, die laut den ermittelten Pfaden auf einem Pfad liegen, der zu dem angestrebten oberen Typ der Typhierarchie führt. Aus der Menge der Features werden diejenigen Feature herausgefiltert, die die Information enthalten, dass der untere Typ den nächsthöheren Typ der Typhierarchie erweitert oder implementiert.

Bei einem Zugriff ohne qualifizierenden Vorsatz ist es nötig herauszufinden, ob die Subtypinformation im Quellfeature dieses Zugriffs fehlt. Ist dies der Fall, so wird mit `addSplError` ermittelt, ob sich aus der Auswahl dieses Quellfeatures auch die Auswahl mindestens eines Features, das die Subtypinformation hat, logisch folgern lässt. Dabei wird die Information übermittelt, dass die zweite Runde bereits durchlaufen wurde.

[1] Dieser Fall kann aber vorliegen, wenn der qualifizierende Vorsatz den gleichen Typ hat wie der Typ, von dem aus der Zugriff erfolgt, oder ein Obertyp von diesem ist.

Damit stoppt die Typprüfung nach dem Durchlaufen des ersten Schritts der Methode `addSplError`.

Bei einem Zugriff mit einem qualifizierenden Vorsatz wird für alle Features, die zu dem qualifizierten, in der Typhierarchie unten stehenden Typ[1] beitragen, analysiert, ob die Subtypinformation in diesen Features fehlt. Ist dies der Fall, so wird mit `addSplError` ermittelt, ob sich aus der Auswahl dieses Quellfeatures auch die Auswahl mindestens eines Features, das die Subtypinformation hat, logisch folgern lässt. Dabei wird die Information übermittelt, dass die zweite Runde bereits durchlaufen wurde. Damit stoppt die Typprüfung nach dem Durchlaufen des ersten Schritts der Methode `addSplError`. Als ausgewählte Features werden in diesem Fall das zugreifende Feature und das Feature, auf das zugegriffen wird, betrachtet.

Falls das obere Ende der Typhierarchie noch nicht erreicht wurde, wird mit dem nächsten Schritt nach oben weiterverfahren. Dies erfolgt rekursiv, dabei kann es mehrere Pfade (Abzweigungen nach oben) geben. Die Rekursion stoppt, wenn der oberste Typ in einem direkten Schritt erreicht werden kann.

**Typhierarchie-Überprüfung** Mit der Methode `TypeDecl.isInTypeHierarchy` wird ermittelt, ob sich ein Typ `Bottom` in der Typhierarchie unterhalb eines anderen Typs `Top` befindet, d.h. ob `Bottom` ein direkter oder indirekter Subtyp des Typs `Top` ist (transitiv über die Typhierarchie).

Hierzu werden die Typen ermittelt, die `Bottom` erweitert oder implementiert. Wenn sich unter diesen Supertypen der Typ `Top` befindet, dann handelt es sich um eine direkte Subtyp-Beziehung. Andernfalls startet die Suche rekursiv bei den Supertypen von `Bottom`.

Es ist ausreichend, wenn mindestens ein Pfad über die Typhierarchie von `Bottom` nach `Top` gefunden wird.

Diese Methode wird benötigt, um herauszufinden, ob es sich bei einem expliziten Cast um einen Up- oder Downcast handelt.

**Ermitteln der Pfade in der Typhierarchie** In der Methode `ASTNode.getTypeHierarchyPaths` werden die Pfade in der Typhierarchie von einem Typ `bottomType` zu einem Typ `topType` mittels einer `HashMap` gespeichert.

In dieser `HashMap` werden für einen Typ die Supertypen gespeichert, die auf einem kürzesten Pfad zum `topType` führen.

Hierzu werden die Typen ermittelt, die `bottomType` erweitert oder implementiert. Wenn sich unter diesen Supertypen der Typ `topType` befindet, dann wird dieser Pfad gespeichert und die Ermittlung der Pfade abgebrochen. Andernfalls startet die Suche rekursiv bei den Supertypen von `bottomType`. Der erste Schritt eines Pfades wird erst dann gespeichert, wenn der restliche Pfad zum Ziel führt. In diesem Fall werden die in weiteren Schritten ermittelten Pfade übernommen.

[1] Dabei ist es wichtig, den genauesten Typ zu ermitteln, der bei der Deklaration der qualifizierenden Variablen oder des qualifizierenden Feldes angegeben wurde.

Es werden nicht alle möglichen Pfade ermittelt, sondern bei einem Fund eines kürzesten Pfades abgebrochen, da ansonsten Probleme durch lange Pfade in der Typhierarchie enstehen könnten, die aus alternativen Implementierungen resultieren. Diese längeren Pfade waren wahrscheinlich *nicht* von der Erstellerin der Softwareproduktlinie beabsichtigt[1].

### 5.1.4 Überprüfung der einzelnen Zugriffe

**Typcheck-Platzhalter in `ASTNode`** Diese Methode dient als Platzhalter für die speziellen, im Nachfolgenden erläuterten `splTypeCheck`s in den Knoten des *Abstract Syntax Tree*, die diese Methode überschreiben.

```
1 /* placeholder */
2 public void ASTNode.splTypeCheck() { }
```

**`VarAccess.splTypeCheck()`** In dieser Methode werden Zugriffe auf Variablen, die genauer betrachtet Zugriffe auf Felder[2] sind, gesammelt und zur Auswertung an die Methode `addSplErrors` übergeben. Das Feld, auf das zugegriffen wird, muss in mindestens einem Feature deklariert werden, das bei Auswahl des zugreifenden Features laut den Abhängigkeiten des Feature-Modells immer verfügbar ist. Die im *Abstract Syntax Tree* gefundenen Ziele für den zu analysierenden Zugriff werden zusammen mit den Typen, in denen diese Ziele liegen, in einer `HashMap` gespeichert.

Die Ermittlung der die Ziele beinhaltenden Typen findet wie auch in allen nachfolgenden `splTypeCheck`-Methoden in dieser Methode statt, da die Ziele in der Methode `addSplErrors` der Klasse `ASTNode` nur als `ASTNode`s gespeichert werden, und für diesen Typ die Methode `hostType()` nicht zur Verfügung steht. Es müsste in einer Fallunterscheidung für jeden möglichen Typ der Ziele diese Methode auf dem genauen Typ aufgerufen werden. Die die Ziele beinhaltenden Typen werden für die Ausgabe der Fehlermeldungen benötigt.

**`MethodAccess.splTypeCheck()`** In dieser Methode werden Zugriffe auf Methoden ausgewertet. Dabei wird zunächst der Aufruf der speziellen Methode `original` behandelt, die bei einer Verfeinerung einer Methode die ursprüngliche Methode aufruft. Dazu muss diese Methode mit der gleichen Signatur wie die Methode, in der dieser `original`-Aufruf stattfindet, in einem anderen Feature stets verfügbar sein. Hierfür werden die Zugriffe auf Methoden mit der gleichen Signatur in einem anderen Feature in diesem Typ gesammelt und ausgewertet. Desweiteren wird für diesen Zugriff auf eine Methode überprüft, ob bei unterschiedlichem Typ der übergebenen Argumente und der erwarteten Parameter die Information, dass der Typ des übergebenen Arguments ein Subtyp des Typs des erwarteten Parameters ist, stets vorhanden ist.

[1] An dieser Stelle wird bereits das Feld der Spekulation betreten, was die Erstellerin bei alternativen Typhierarchien im Sinn hatte. Für die Typprüfung muss man sich an dieser Stelle für eine Typhierarchie entscheiden. Das ist hier diejenige mit den kürzesten Pfaden.

[2] Eine eigene Klasse `FieldAccess` sieht JastAddJ nicht vor

Danach werden die Methoden-Aufrufe, die sich nicht auf `original` beziehen, ausgewertet.

Auch hier findet wieder die Überprüfung statt, ob bei unterschiedlichem Typ der übergebenen Argumente und der erwarteten Parameter die Information, dass der Typ des übergebenen Arguments ein Subtyp des Typs des erwarteten Parameters ist, stets vorhanden ist.

**`TypeAccess.splTypeCheck()`** Die Zugriffe auf Typen werden in einer Datenstruktur gesammelt. Diese Datenstruktur und die Auswertung der Typzugriffe werden im Abschnitt „Sammlung der Typzugriffe“ auf Seite 31 beschrieben. Zugriffe auf Typen aus `import`-Statements werden ignoriert. Aus den `import`-Statements kann das Feature, das dieses `import`-Statement einführt, nicht ermittelt werden, da die `import`-Statements von Fuji dem ersten Feature zugeordnet werden, das die Klasse einführt.

**`ClassInstanceExpr.splTypeCheck()`** In dieser Methode werden Aufrufe eines Konstruktors ausgewertet.

Desweiteren wird für diesen Konstruktor überprüft, ob bei unterschiedlichem Typ der übergebenen Argumente und der erwarteten Parameter die Information, dass der Typ des übergebenen Arguments ein Subtyp des Typs des erwarteten Parameters ist, stets vorhanden ist.

**`FieldDeclaration.splTypeCheck()`** Die Felddeklarationen, bei denen ein impliziter Cast stattfindet, werden ausgewertet.

Dazu wird aus der Definition der rechten Seite der Zuweisung, falls diese existiert, der Typ dieser Definition ermittelt und, wenn dieser nicht mit dem Typ der Felddeklaration übereinstimmt, ausgewertet.

**`VariableDeclaration.splTypeCheck()`** Die Variablendeklarationen, bei denen ein impliziter Cast stattfindet, werden ausgewertet.

Dazu wird aus der Definition der rechten Seite der Zuweisung, falls diese existiert, der Typ dieser Definition ermittelt und, wenn dieser nicht mit dem Typ der Variablendeklaraton übereinstimmt, ausgewertet.

**`AssignSimpleExpr.splTypeCheck()`** Die Zuweisungen, die keine Deklarationen sind, bei denen ein impliziter Cast stattfindet, werden ausgewertet.

Dazu wird aus der rechten Seite der Zuweisung, falls diese existiert, der Typ dieser Definition ermittelt und, wenn dieser nicht mit dem Typ der Zuweisung übereinstimmt, ausgewertet.

**`CastExpr.splTypeCheck()`** Die expliziten Casts, die Up- oder Downcasts sein können, werden ausgewertet.

Wenn es sich um unterschiedliche Typen handelt, dann wird nach Up- oder Downcast unterschieden, um diesen Zugriff auszuwerten.

`SuperConstructorAccess.splTypeCheck()` Die Zugriffe auf Konstruktoren der Oberklasse werden ausgewertet.

Außerdem findet bei einem Aufruf von `super` eine Überprüfung statt, ob die Information, dass bei unterschiedlichem Typ des übergebenen Arguments und des erwarteten Parameters der Typ des übergebenen Arguments ein Subtyp des Typs des erwarteten Parameters ist, immer vorhanden ist.

**Sammlung der Typzugriffe** Alle Zugriffe auf Typen werden in einer `HashMap` namens `ASTNode.typeAccesses` gespeichert. Hierin werden Zugriffe auf Typen eines bestimmten Quellfeatures in einem bestimmten Quelltyp auf einen bestimmten Zieltyp gesammelt. Diese Zugriffe werden in der Klasse `TypeErrorInfo` (siehe Abschnitt 5.3) zusammen mit den jeweiligen Dateinamen und Zeilen- und Spaltenummern, aus denen die Zugriffe stammen, gespeichert.

Die Auswertung findet nach dem Durchlaufen des *Abstract Syntax Tree*s für diese gesammelten Zugriffe statt.

**Einleiten der Überprüfung der Typzugriffe** In der Methode `ASTNode.collectSplTypeErrors` werden alle Typzugriffe ausgewertet. Dazu wird die Sammlung der Zugriffe anhand der Quellfeature-IDs (d.h. alle Features, die Zugriffe auf Typen haben) durchgegangen. Für jedes solche Quellfeature werden die die Zugriffe umgebenden Typen durchgegangen, und für diese die die Ziele umgebenden Typen. Für diese Typen werden die Zugriffe ermittelt, diese zusammen mit den die Ziele umgebenden Typen in einer `HashMap` zwischengespeichert und diese Zugriffe ausgewertet.

## 5.2 Modifikationen in der Datei `fuji/Main.java` von Fuji

Für die Typprüfung wurden in der Hauptdatei `Main.java` des Fuji-Compilers einige Änderungen durchgeführt.

Eine neue Option namens `-typechecker` wurde für den Aufruf von Fuji auf der Konsole hinzugefügt. Das Programm erwartet nach Auswahl dieser Option als Parameter nicht mehr eine Datei mit der Liste der zu komponierenden Features, sondern ein Feature-Modell in Form einer Datei im GUIDSL-Format[1]. Diese Datei wird eingelesen und als `FeatureModel` gespeichert.

Diese Option muss zusammen mit der bestehenden Option `-fopRefs` gewählt werden, die die Referenzen zwischen allen Features der Produktlinie ermittelt und dabei eine andere Kompositionsstrategie als bei der Komposition eines Produkts verwendet, um alternative Implementierungen zu berücksichtigen [Kol11]. Durch die Auswahl dieser beiden Optionen wird nicht wie sonst die Kompilierung einer Variante anhand einer Liste von Features durchgeführt, sondern mithilfe des Feature-Modells eine Typprüfung von oben nach unten im *Abstract Syntax Tree* ausgelöst. Dieser *Abstract Syntax Tree* wird dabei aus allen konkreten Features des Feature-Modells erzeugt.

Danach wird die Typprüfung auf dem obersten Knoten des *Abstract Syntax Tree*s (eine Instanz der Klasse `Program`) gestartet, der dazu das Feature-Modell, die Liste der Fehlermeldungen und die Liste der Warnungen mitgegeben werden.

```
1  if (!cmd.hasOption(PROG_MODE)) {
2    Composition composition = new Composition(this);
3    if (cmd.hasOption(TYPECHECKER)) {
4      Iterator<Program> astIter = composition.getASTIterator();
5      Program ast = astIter.next();
6      ast.splErrorCheck(model, errors, warnings);
7    } else {
8      processAST(composition);
9    }
10   /* ... */
11 }
```

Anschließend werden die Fehlermeldungen bzw. die Compiler-Warnungen ausgegeben.

Für die Zeitmessung der Typprüfung wurden weitere Änderungen durchgeführt, siehe dazu den Abschnitt 6.13.1 „Änderungen an Main.java für die Zeitmessung“ auf Seite 59.

[1] `http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/guidsl.html`, besucht am 2013-04-09

## 5.3 Die Hilfsdatei `fuji/TypeErrorInfo.java`

Damit nicht für jeden fehlerhaften Typzugriff eine eigene Fehlermeldung ausgeben wird, wenn der Typ und das Feature, aus denen heraus der Zugriff erfolgt, und die Ziele die gleichen sind, werden diese Zugriffe gesammelt ausgegeben. Die einzelnen Fehlermeldungen würden sich nur in der Zeilen- und Spaltennummer unterscheiden. Dadurch werden weniger fast gleiche Fehlermeldungen ausgegeben, wodurch die Übersichtlichkeit der Fehlerausgabe etwas verbessert wird.

Dazu ist es nötig, die Dateinamen, Zeilen- und Spaltennummern zu speichern, um bei der Prüfung dieser Zugriffe die korrekten Dateinamen, Zeilen- und Spaltennummern zusammen mit der eigentlichen Fehlermeldung ausgeben zu können[1]. Diese Angaben werden zusammen mit den Zielen für diesen Zugriff in dieser Klasse gespeichert. Außerdem wird in dieser Klasse gespeichert, ob es sich bei dem Zugriff um einen Zugriff mit qualifizierendem Vorsatz handelt, der angibt, wo das Ziel dieses Zugriffs zu finden ist.

[1] Ansonsten würden die Dateinamen, Zeilen- und Spaltennummern beim Aufruf der Methode, die die Fehlermeldung erzeugt, ermittelt. Dies würde bei der hier stattfindenen Erzeugung der Fehlermeldungen nach dem Durchlaufen des *Abstract Syntax Tree* zu fehlerhaften Angaben führen.

# 6 Evaluierung

Aus der Menge an Softwareproduktlinien aus dem Beispiel-Ordner von Fuji wurden folgende zwölf Softwareproduktlinien ausgewählt. Dabei finden sich einige kleine, ein paar mittelgroße und mit Violet auch eine Softwareproduktlinie mit vielen Features.

Für diese Softwareproduktlinien wurde die Typprüfung mit dem entwickelten Typchecker durchgeführt und die auftretenden Fehlermeldungen analysiert. Aus diesen Fehlermeldungen lassen sich dann Constraints ableiten, die, wenn sie zum Feature-Modell hinzugefügt werden, diese Typfehler korrigieren.

| SPL | Anzahl Features | Lines of Code | Anzahl Produkte |
|---|---:|---:|---:|
| EPL | 12 | 111 | 425 |
| GPL | 27 | 1930 | 156 |
| Graph | 5 | 251 | 16 |
| GUIDSL | 26 | 10084 | 24 |
| Notepad | 12 | 937 | 512 |
| PKJab | 8 | 3373 | 48 |
| Prevayler | 6 | 5268 | 32 |
| Raroscope | 5 | 316 | 16 |
| Sudoku | 7 | 1422 | 64 |
| TankWar | 37 | 4845 | 2458 |
| Violet | 88 | 7151 | — |
| ZipMe | 13 | 3446 | 24 |

Tabelle 2: Übersicht über die Softwareproduktlinien für die Evaluierung

Die *Lines of Code* wurden ohne Kommentare gemessen. Für Violet war es nicht möglich, die Anzahl der validen Produkte dieser Produktlinie zu ermitteln. Die SAT-Solver `sat4j`[1] und `CHOCO`[2] brachen mit `OutOfMemory`-Exceptions ab.

Die Pfade in den Fehlermeldungen wurden gekürzt.

[1] `http://www.sat4j.org/`, besucht am 2013-04-21
[2] `http://www.emn.fr/z-info/choco-solver/`, besucht am 2013-04-21

## 6.1 EPL

Das Feature-Modell der Softwareproduktlinie **EPL** (eine feature-orientierte Java-Implementierung des *Expression Problem*s, [LHB04]) ist in Abbildung 3 auf Seite 15 zu sehen.

### Analyse der Fehlerausgabe

Exemplarisch werden hier die Fehlermeldungen aus dem Motivations-Beispiel auf Seite 14 analysiert:

```
.../EPL/features/Plus_ToString/tmp/Plus.java:5,12:
  Semantic Error: MAYBE dependency:
Feature Plus_ToString accesses the field
  Exp x;
of feature Plus.
Feature Plus may not be present in every valid selection.

.../EPL/features/Plus_ToString/tmp/Plus.java:5,24:
  Semantic Error: MAYBE dependency:
Feature Plus_ToString accesses the field
  Exp y;
of feature Plus.
Feature Plus may not be present in every valid selection.
```

Die Klasse `tmp.Plus` des Features PLUS_TOSTRING greift in Zeile 5 auf die Felder `Exp x;` und `Exp y;` der Klasse `tmp.Plus` des Features PLUS zu. Das Feature PLUS_TOSTRING verfeinert die Klasse `tmp.Plus` des Features PLUS durch das Hinzufügen einer neuen Methode `toString()`. Die bei der Verfeinerung benötigten Felder `Exp x;` und `Exp y;` sind innerhalb der Klasse `tmp.Plus` nur im Feature PLUS vorhanden. Wenn das Feature PLUS_TOSTRING ausgewählt wurde, ist nicht sichergestellt, dass auch das Feature PLUS ausgewählt wurde, da das Feature PLUS im Feature-Modell als optional gekennzeichnet ist und kein Constraint im Feature-Modell vorhanden ist, das aufgrund der Auswahl des Features PLUS_TOSTRING die Auswahl des Features PLUS mittels einer Implikation PLUS_TOSTRING $\Rightarrow$ PLUS erzwingen würde. Diese *maybe*-Beziehung der beiden Features resultiert in einem Typfehler, da es somit laut Feature-Modell valide Produkte geben kann, die einen Kompilerfehler erzeugen, da der Feldzugriff ins Leere geht, wenn das Zielfeature nicht ausgewählt wurde.

Eine komplette Liste der Fehlermeldungen und eine komplette Analyse der Fehlermeldungen ist auf der beiliegenden CD zu finden.

Für die Zugriffe ist zusammenfassend zu sagen, dass es möglich ist, ein *DERIVATIVE* auszuwählen, ohne die zugehörige *OPERATION* oder *STRUCTURE* auswählen zu müssen. Für jede Kombination einer *OPERATION* und einer *STRUCTURE* gibt es im Feature-Modell ein Constraint, das das zugehörige *DERIVATIVE* impliziert. Da ein *DERIVATIVE* aber auf seine zugehörige *OPERATION* oder *STRUCTURE* zugreift, sollte es sich bei all diesen Constraints eigentlich um Äquivalenzen handeln[1].

[1] Für eine Diskussion über mögliche Auswahl-Einschränkungen dieser *DERIVATIVES* in einer Benutzeroberfläche siehe Abschnitt 3.1 auf Seite 14

Das Feature Eval benötigt außerdem Zugriff auf Konstruktoren aus den Features Plus und Neg. Auch das Feature ToString benötigt Zugriff auf Konstruktoren aus den Features Plus und Neg.

Außerdem kann es passieren, dass für das Feature Eval und das Feature ToString die Typen `Plus` bzw. `Neg` nicht zur Verfügung stehen. Diese Typen werden zwar jeweils dreimal zur Verfügung gestellt, allerdings befinden sie sich alle in optionalen Features, so das es passieren kann, dass gar kein passender Typ in einem Feature ausgewählt ist.

### False Positives

In weiteren Fehlermeldungen wird das Interface `Exp` als optional gekennzeichet, obwohl es im Feature base immer zur Verfügung steht. Dies liegt daran, dass das Interface `Exp` im Feature base ohne jeglichen Inhalt implementiert wird. Der Typchecker ist aber so implementiert, dass zur Verfügung stehende Typen in bestimmten Features anhand des Vorhandensein von mindestens einer Struktur im Rumpf dieses Typs erkannt werden. Deshalb wird dieser leere Interface-Rumpf nicht als verfügbar erkannt und es werden fälschlicherweise Fehlermeldungen ausgegeben. Dieses Problem wird im Abschnitt 7.1 auf Seite 62 diskutiert.

Die folgenden Fehlermeldungen beziehen sich alle auf dieses scheinbare Nichtvorhandensein des Interfaces `Exp` im Feature base.

```
.../EPL/features/Neg/tmp/Neg.java:3
.../EPL/features/Neg/tmp/Neg.java:4
.../EPL/features/Neg/tmp/Neg.java:5:
  Semantic Error: 2 optional targets
  (there may be a valid selection where none of these targets is present):
Feature Neg accesses:
- the type
    tmp.Exp
  of feature Eval
- the type
    tmp.Exp
  of feature ToString

.../EPL/features/Plus/tmp/Plus.java:5
.../EPL/features/Plus/tmp/Plus.java:6
.../EPL/features/Plus/tmp/Plus.java:3
.../EPL/features/Plus/tmp/Plus.java:4:
  Semantic Error: 2 optional targets
  (there may be a valid selection where none of these targets is present):
Feature Plus accesses:
- the type
    tmp.Exp
  of feature Eval
- the type
    tmp.Exp
  of feature ToString

.../EPL/features/Numbers/tmp/Num.java:3:
  Semantic Error: 2 optional targets
  (there may be a valid selection where none of these targets is present):
Feature Numbers accesses:
- the type
    tmp.Exp
```

```
  of feature Eval
- the type
    tmp.Exp
  of feature ToString

.../EPL/features/base/tmp/Test.java:4:
  Semantic Error: 2 optional targets
  (there may be a valid selection where none of these targets is present):
Feature base accesses:
- the type
    tmp.Exp
  of feature Eval
- the type
    tmp.Exp
  of feature ToString
```

### Notwendige Änderungen im Feature-Modell, um diese Fehler zu korrigieren

Diese Fehler können nun dadurch korrigieren werden, dass folgende Constraints zum Feature-Modell hinzugefügt werden. Somit wird sichergestellt, dass für die Zugriffe benötigte Ziele in jedem Produkt vorhanden sind, in dem die zugreifenden Features vorhanden sind. Allerdings bleiben die *false positive*-Fehlermeldungen weiterhin bestehen.

- Eval_Neg ⇒ Eval ∧ Neg
- Neg_ToString ⇒ Neg
- Eval_Plus ⇒ Eval ∧ Plus
- Plus_ToString ⇒ Plus
- Eval ⇒ Plus ∧ Neg
- ToString ⇒ Plus ∧ Neg

An dieser Stelle fällt auf, dass die *Derivatives* mit „ToString“ im Namen scheinbar keinen Zugriff auf das Feature ToString benötigen. Die in diesen *Derivatives* referenzierte `toString`-Methode ist in Java bereits in der Klasse `Object` vorhanden und wird im Feature ToString überschrieben. Wenn die überschriebene `toString`-Methode nicht zur Verfügung steht, geht der Zugriff auf die Klasse `Object`. Dadurch kommt es zu nicht beabsichtigten Ausgaben. Diese Situation kann der implementierte Typchecker nicht erkennen, da die `toString`-Methode in der Klasse `Object` als ein mögliches Ziel immer zur Verfügung steht.

## 6.2 GPL

Das Feature-Modell der Softwareproduktlinie **GPL** (*Graph Product Line*, eine Familie klassischer Graph-Anwendungen [LHB01]) ist in Abbildung 4 zu sehen.

### Analyse der Fehlerausgabe:

An dieser Stelle wird eine Fehlermeldung exemplarisch analysiert:

```
.../GPL/Graph.java:22,38:
  Semantic Error: 2 optional targets
  (there may be a valid selection where none of these targets is present):
Feature WeightedWithNeighbors accesses:
- the field
    public Vertex neighbor;
  of feature DirectedWithNeighbors
- the field
    public Vertex neighbor;
  of feature UndirectedWithNeighbors
```

Die Klasse `GPL.Graph` des Features WeightedWithNeighbors benötigt in Zeile 22 Zugriff auf das Feld `public Vertex neighbor;` der Klasse `GPL.Neighbor`. Dieses Feld wird von den Features DirectedWithNeighbors und UndirectedWithNeighbors jeweils in der Klasse `GPL.Neighbor` bereitgestellt. Wenn das Feature WeightedWithNeighbors ausgewählt wurde, ist nicht sichergestellt, dass dann auch wenigstens eines der Features DirectedWithNeighbors und UndirectedWithNeighbors ausgewählt wurde, die diese Klasse definieren oder verfeinern, da alle diese Zielfeatures im Feature-Modell als optional gekennzeichnet sind und kein Constraint im Feature-Modell vorhanden ist, das aufgrund der Auswahl des Features WeightedWithNeighbors die Auswahl wenigstens eines der Features DirectedWithNeighbors und UndirectedWithNeighbors mittels einer Implikation WeightedWithNeighbors $\Rightarrow$ wenigstens ein Feature aus {DirectedWithNeighbors und UndirectedWithNeighbors} erzwingen würde. Diese *maybe*-Beziehungen des Quellfeatures zu den Zielfeatures resultieren in einem Typfehler, da es somit laut Feature-Modell valide Produkte geben kann, die einen Kompilerfehler erzeugen, da der Feldzugriff ins Leere geht, wenn keines der Zielfeatures ausgewählt wurde.

Eine komplette Liste der Fehlermeldungen und eine komplette Analyse der Fehlermeldungen ist auf der beiliegenden CD zu finden.

Die drei Features WeightedWithNeighbors, WeightedOnlyVertices und WeightedWithEdges benötigen jeweils entweder die zugehörige Directed- oder Undirected-Variante von -WithNeighbors, -OnlyVertices bzw. -WithEdges. Es ist aber möglich, dass keine dieser Varianten ausgewählt wurde.

Diese Situation ist ähnlich der Situation in der Softwareproduktlinie EPL. Auch hier werden mittels Constraints bei einer Kombination von Features die entsprechenden Features für die Kombination impliziert, aus einer Auswahl eines Kombinations-Features

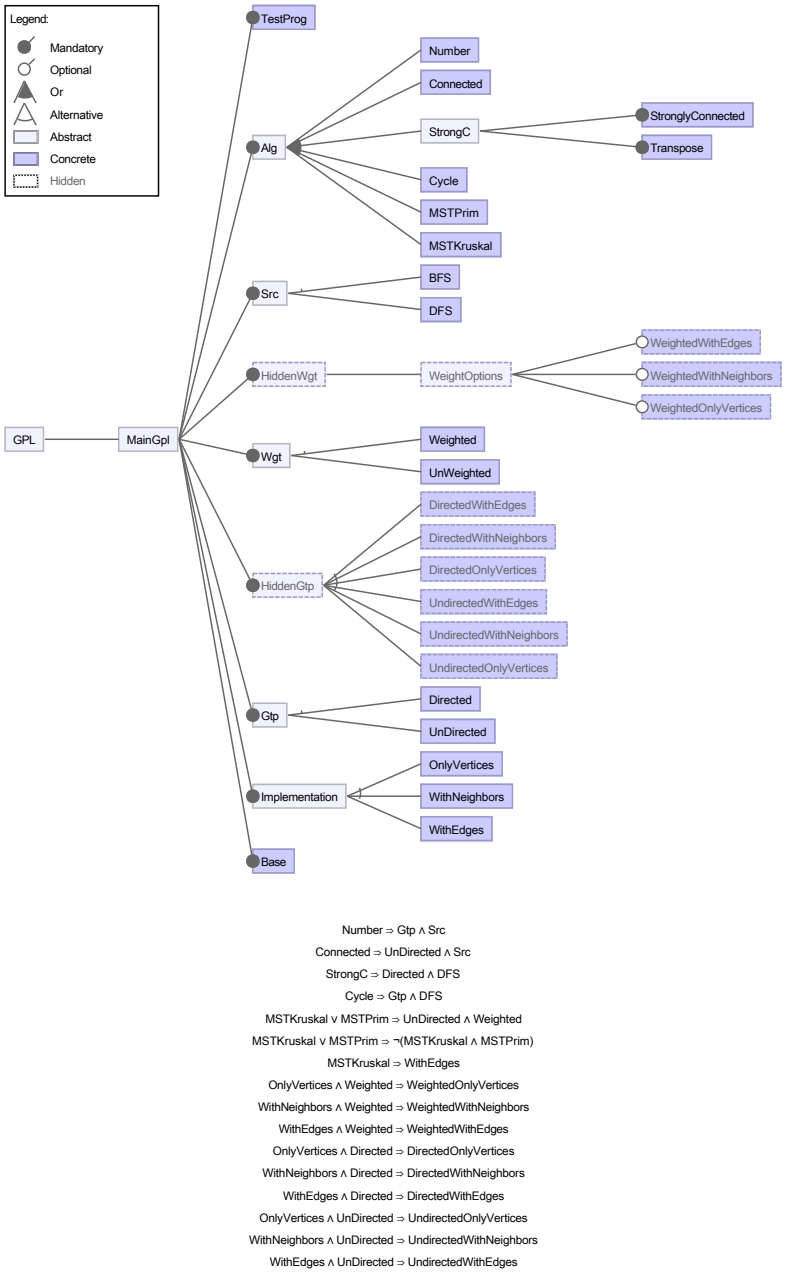



Abbildung 4: GPL-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13]).
Die Kennzeichnung mancher Alternativ-Gruppen ist leider nur undeutlich zu sehen (angedeutete vertikale Striche), bedingt durch das gewählte Layout aber nicht besser darstellbar.

folgt allerdings nicht die Auswahl eines zugehörigen Basis-Features dieser Kombination. Diese problematischen Features sind im Feature-Modell von GPL aber als *hidden* annotiert, so dass sie bei der Konfiguration eines Produkts nicht direkt ausgewählt werden können[1]. Damit wird dann aber die Typsicherheit auf der richtigen Umsetzung der *hidden*-Annotation (etwa für eine graphische Benutzeroberfläche) aufgebaut.

#### Notwendige Änderungen im Feature-Modell, um diese Fehler zu korrigieren

Diese Fehler können nun dadurch korrigieren werden, dass folgende Constraints zum Feature-Modell hinzugefügt werden. Somit wird sichergestellt, dass für die Zugriffe benötigte Ziele in jedem Produkt vorhanden sind, in dem die zugreifenden Features vorhanden sind.

- WeightedOnlyVertices ∧ Directed ⇒ DirectedOnlyVertices
- WeightedOnlyVertices ∧ UnDirected ⇒ UndirectedOnlyVertices
- WeightedWithEdges ∧ Directed ⇒ DirectedWithEdges
- WeightedWithEdges ∧ UnDirected ⇒ UndirectedWithEdges
- WeightedWithNeighbors ∧ Directed ⇒ DirectedWithNeighbors
- WeightedWithNeighbors ∧ UnDirected ⇒ UndirectedWithNeighbors

## 6.3 Graph

Für die Softwareproduktlinie **Graph** lag kein Feature-Modell vor. Es wurde angenommen, dass alle Features bis auf das Basis-Feature BasicGraph optional sind. Dieses erzeugte Feature-Modell ist in Abbildung 5 zu sehen.

Bei der Typprüfung ergaben sich keine Fehlermeldungen. Auch bei der Komposition und der anschließenden Kompilierung der einzelnen 16 Varianten mit Fuji ergaben sich keine Fehlermeldungen.

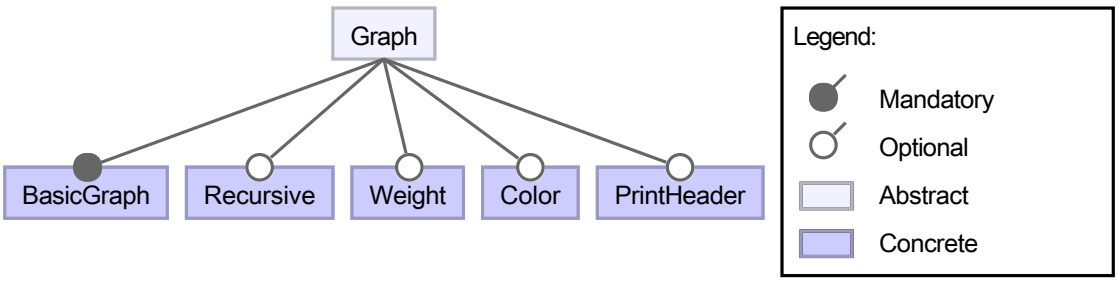



Abbildung 5: Graph-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13])

[1] `http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/guidsl.html#Annotations`, besucht am 2013-04-22

## 6.4 GUIDSL

Für die Softwareproduktlinie **GUIDSL** lag kein Feature-Modell vor. GUIDSL ist ein Werkzeug zur Konfiguration von Produktlinien. Feature-Modelle im GUIDSL-Format können eingelesen werden. Die verschiedenen Produkte dieser Produktlinie können anhand des eingelesenen Feature-Modells konfiguriert und analysiert werden [Kol11].

Es wurde zunächst angenommen, dass die Features in Form eines *Stack*s aufeinander aufbauen, d.h. alle Features bis auf das Basis-Feature KERNEL werden als optional angenommen und durch Constraints wird erreicht, dass die Liste der Features von oben nach unten wächst, indem jedes neu hinzugenommene Feature das vorhergehende Feature impliziert. Allerdings zeigte sich, dass das Feature KERNEL mehrfach auf das Feature GENBALI zugreift und das Feature GENBALI massiv auf das Feature DGRAM zugreift. Deshalb wurde außerdem die Annahme getroffen, dass die ersten drei Features obligatorisch sind. Dieses erzeugte Feature-Modell ist in Abbildung 6 zu sehen.

### Analyse der Fehlerausgabe

An dieser Stelle wird eine Fehlermeldung exemplarisch analysiert:

```
.../GUIDSL/features/eharvest/BAnd.java:6:
  Semantic Error: MAYBE dependency:
Feature eharvest accesses the constructor
  public and(node l, node r) { ... }
of feature bexpr.
Feature bexpr may not be present in every valid selection.
```

Die Klasse `BAnd.java` des Features EHARVEST greift in Zeile 6 auf den Konstruktor `and(node l, node r)` der Klasse `and.java` des Features BEXPR zu. Der benötigte Konstruktor `and(node l, node r)` ist innerhalb der Klasse `and.java` nur im Feature BEXPR vorhanden. Wenn das Feature EHARVEST ausgewählt wurde, ist nicht sichergestellt, dass auch das Feature BEXPR ausgewählt wurde, da das Feature BEXPR im Feature-Modell als optional gekennzeichnet ist und kein Constraint im Feature-Modell vorhanden ist, das aufgrund der Auswahl des Features EHARVEST die Auswahl des Features BEXPR mittels einer Implikation $\text{EHARVEST} \Rightarrow \text{BEXPR}$ erzwingen würde. Diese *maybe*-Beziehung der beiden Features resultiert in einem Typfehler, da es somit laut Feature-Modell valide Produkte geben kann, die einen Kompilerfehler erzeugen, da der Konstruktorzugriff ins Leere geht, wenn das Zielfeature nicht ausgewählt wurde.

Eine komplette Liste der Fehlermeldungen und eine komplette Analyse der Fehlermeldungen ist auf der beiliegenden CD zu finden.

### Notwendige Änderungen im Feature-Modell, um diese Fehler zu korrigieren

Diese Fehler können nun dadurch korrigieren werden, dass folgende Constraints zum Feature-Modell hinzugefügt werden. Somit wird sichergestellt, dass für die Zugriffe benötigte Ziele in jedem Produkt vorhanden sind, in dem die zugreifenden Features vorhanden sind.

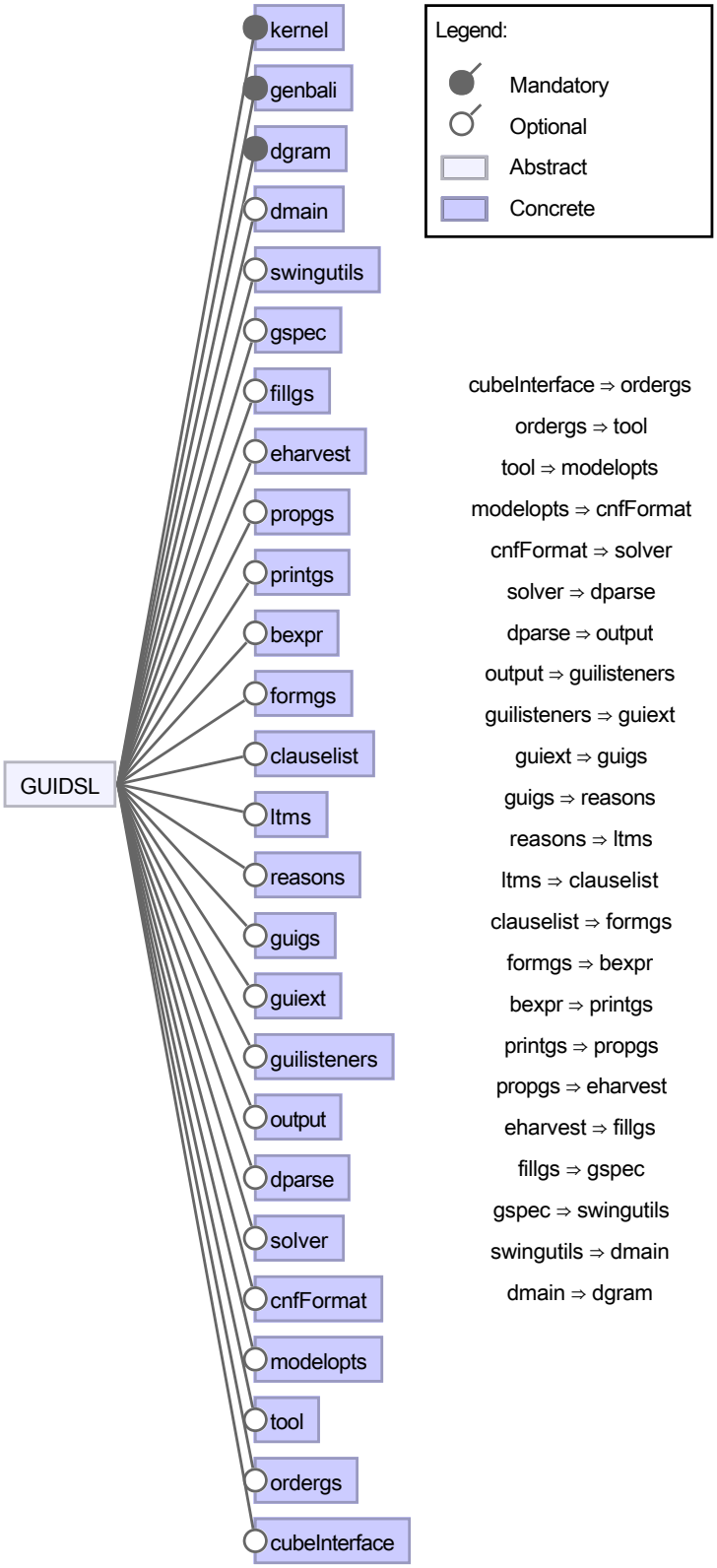



Abbildung 6: GUIDSL-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13])

- EHARVEST ⇒ BEXPR ∧ CNFFORMAT
- CLAUSELIST ⇒ CNFFORMAT
- GUIGS ⇒ CNFFORMAT ∧ GUIEXT ∧ GUILISTENERS
- GUILISTENERS ⇒ OUTPUT

## 6.5 Notepad

Das Feature-Modell der Softwareproduktlinie **Notepad** ist in Abbildung 7 zu sehen.

Notepad ist ein Texteditor, dessen Varianten von einem rudimentären Notizblock ohne Speichermöglichkeit bis zu einem funktionell reichhaltigen Editor mit verschiedenen Schriftauszeichnungsmöglichkeiten reichen [Kol11].

Bei der Typprüfung ergaben sich keine Fehlermeldungen. Bei der Komposition und der anschließenden Kompilierung mit Fuji ergab sich in den einzelnen 512 Varianten nur jeweils eine *Deprecated*-Warnung.

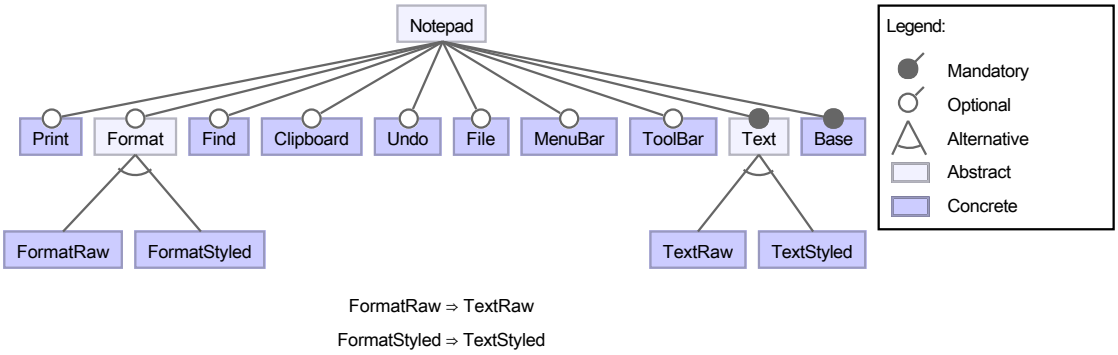



Abbildung 7: Notepad-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13])

## 6.6 PKJab

Das Feature-Modell der Softwareproduktlinie **PKJab** ist in Abbildung 8 zu sehen.

PKJab ist ein XMPP-Instant-Messaging-Client. XMPP steht für *Extensible Messaging and Presence Protocol*, das auch als *Jabber*-Protokoll bekannt ist [Kol11].

Bei der Typprüfung ergaben sich keine Fehlermeldungen. Auch bei der Komposition und der anschließenden Kompilierung mit Fuji ergaben sich in den einzelnen 48 Varianten keine Fehlermeldungen.

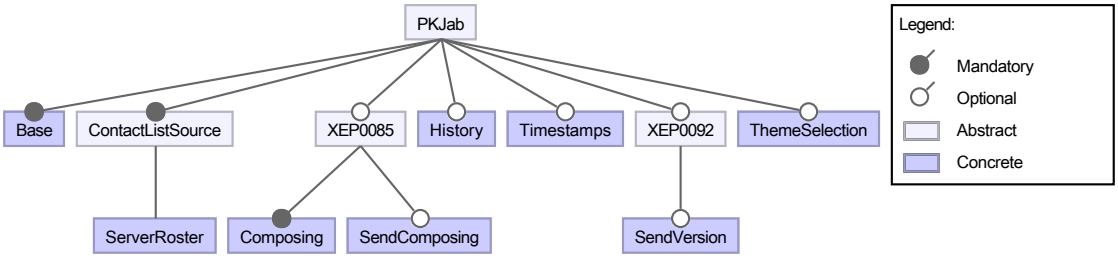



Abbildung 8: PKJab-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13])

## 6.7 Prevayler

Das Feature-Modell der Softwareproduktlinie **Prevayler** ist in Abbildung 9 zu sehen. Prevayler ist eine Open-Source-Bibliothek für Objektpersistenz in Java [Kol11].

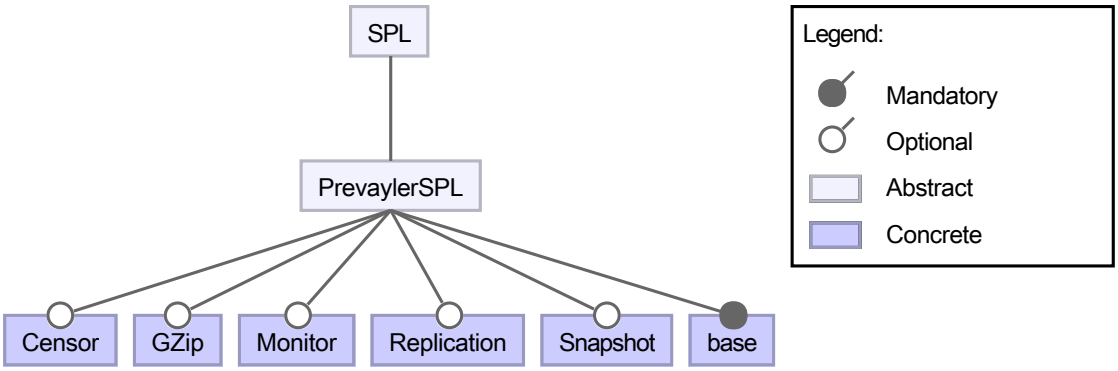



Abbildung 9: Prevayler-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13])

### Analyse der Fehlerausgabe

An dieser Stelle wird eine Fehlermeldung exemplarisch analysiert:

```
.../Prevayler/features/Snapshot/org/prevayler/PrevaylerFactory.java:46,11:
  Semantic Error: MAYBE dependency:
Feature Snapshot accesses the field
  private String _remoteServerIpAddress;
of feature Replication.
Feature Replication may not be present in every valid selection.
```

Die Klasse `org.prevayler.PrevaylerFactory` des Features Snapshot greift in Zeile 46 auf das Feld `private String _remoteServerIpAddress;` der Klasse `org.prevayler.PrevaylerFactory` des Features Replication zu. Das Feature Snapshot verfeinert die Klasse `org.prevayler.PrevaylerFactory` des Features Replication durch Hinzufügen der Methode `private TransactionPublisher publisher(GenericSnapshotManager snapshotManager) throws IOException`. Das bei der Verfeinerung benötigte Feld `private String _remoteServerIpAddress;` ist innerhalb der Klasse `org.prevayler.PrevaylerFactory` nur im Feature Replication vorhanden. Wenn das Feature Snapshot ausgewählt wurde, ist nicht sichergestellt, dass auch das Feature Replication ausgewählt wurde, da das Feature Replication im Feature-Modell als optional gekennzeichnet ist und kein Constraint im Feature-Modell vorhanden ist, das aufgrund der Auswahl des Features Snapshot die Auswahl des Features Replication mittels einer Implikation Snapshot $\Rightarrow$ Replication erzwingen würde. Diese *maybe*-Beziehung der beiden Features resultiert in einem Typfehler, da es somit laut Feature-Modell valide Produkte geben kann, die einen Kompilerfehler erzeugen, da der Feldzugriff ins Leere geht, wenn das Zielfeature nicht ausgewählt wurde.

Eine komplette Liste der Fehlermeldungen und eine komplette Analyse der Fehlermeldungen ist auf der beiliegenden CD zu finden.

Das Feature BASE benötigt das Feature SNAPSHOT. Da BASE das Basis-Feature ist, das in jedem Produkt vorhanden sein muss, dürfte deshalb gar kein Produkt fehlerfrei kompilieren. Das wird auch dadurch deutlich, dass die Klasse `prevayler.implementation.PrevaylerImpl` im Feature BASE die drei Methoden:

- `takeSnapshot()`
- `prevalentSystem()`
- `execute(org.prevayler.Query)`

aus dem Interface `org.prevayler.Prevayler` nicht implementiert, die erst im Feature SNAPSHOT implementiert werden.

#### Notwendige Änderungen im Feature-Modell, um diese Fehler zu korrigieren

Diese Fehler können nun dadurch korrigieren werden, dass folgende Constraints zum Feature-Modell hinzugefügt werden. Somit wird sichergestellt, dass für die Zugriffe benötigte Ziele in jedem Produkt vorhanden sind, in dem die zugreifenden Features vorhanden sind.

- SNAPSHOT $\Rightarrow$ REPLICATION $\wedge$ CENSOR $\wedge$ MONITOR
- CENSOR $\Rightarrow$ SNAPSHOT
- BASE $\Rightarrow$ SNAPSHOT

Mit diesen zusätzlichen Constraints würde die Anzahl der möglichen Produkte auf zwei zusammenschrumpfen. Nur noch das Feature GZIP wäre optional.

## 6.8 Raroscope

Für die Softwareproduktlinie **Raroscope** lag kein Feature-Modell vor.

Raroscope ist eine Java-Bibliothek für RAR-Archiv-Inhalte[1].

Es wurde angenommen, dass alle Features bis auf das Basis-Feature BASE optional sind. Dieses erzeugte Feature-Modell ist in Abbildung 10 zu sehen. Bei der Typüberprüfung ergaben sich keine Fehlermeldungen. Auch bei der Komposition und der anschließenden Kompilierung mit Fuji ergaben sich in den einzelnen 16 Varianten keine Fehler.

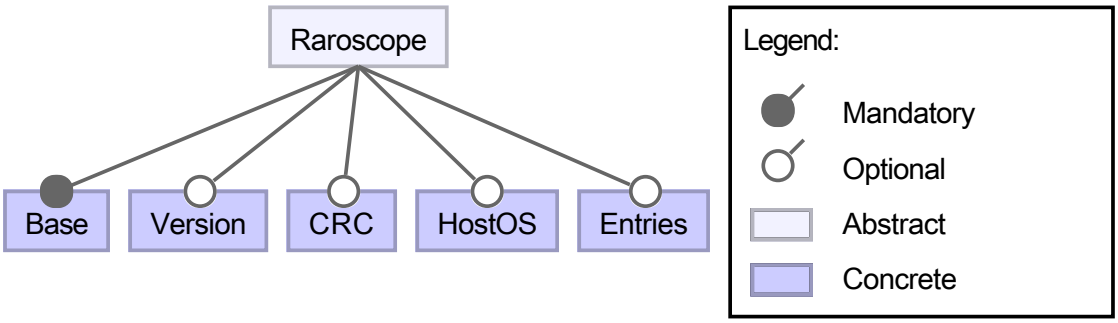



Abbildung 10: Raroscope-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13])

[1] `http://code.google.com/p/raroscope/`, besucht am 2013-04-22

## 6.9 Sudoku

Für die Softwareproduktlinie **Sudoku** lag kein Feature-Modell vor.

Die Produktlinie löst Sudoku-Rätsel.

Es wurde angenommen, dass alle Features bis auf das Basis-Feature BASE optional sind. Dieses erzeugte Feature-Modell ist in Abbildung 11 zu sehen.

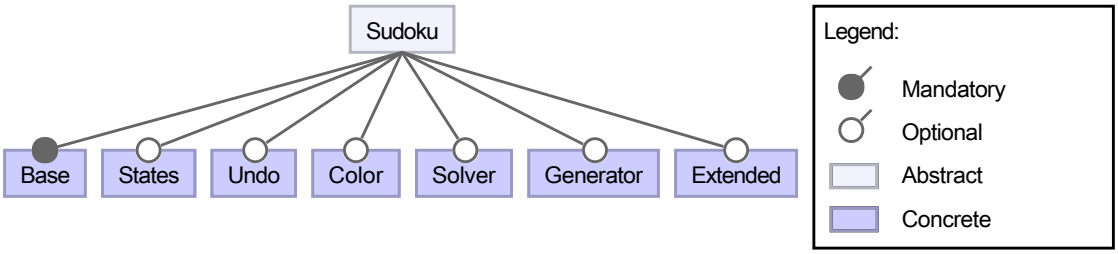



Abbildung 11: Sudoku-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13])

### Analyse der Fehlerausgabe

An dieser Stelle wird eine Fehlermeldung exemplarisch analysiert:

```
.../Sudoku/features/Solver/BoardManager.java:29:
  Semantic Error: MAYBE dependency:
Feature Solver accesses the method
  public void undo() { ... }
of feature Undo.
Feature Undo may not be present in every valid selection.
```

Die Klasse `BoardManager` des Features SOLVER greift in Zeile 29 auf die Methode `public void undo()` der Klasse `BoardManager` des Features UNDO zu. Die benötigte Methode `public void undo()` ist innerhalb der Klasse `BoardManager` nur im Feature UNDO vorhanden. Wenn das Feature SOLVER ausgewählt wurde, ist nicht sichergestellt, dass auch das Feature UNDO ausgewählt wurde, da das Feature UNDO im Feature-Modell als optional gekennzeichnet ist und kein Constraint im Feature-Modell vorhanden ist, das aufgrund der Auswahl des Features SOLVER die Auswahl des Features UNDO mittels einer Implikation SOLVER $\Rightarrow$ UNDO erzwingen würde. Diese *maybe*-Beziehung der beiden Features resultiert in einem Typfehler, da es somit laut Feature-Modell valide Produkte geben kann, die einen Kompilerfehler erzeugen, da der Methodenzugriff ins Leere geht, wenn das Zielfeature nicht ausgewählt wurde.

Eine komplette Liste der Fehlermeldungen und eine komplette Analyse der Fehlermeldungen ist auf der beiliegenden CD zu finden.

#### Notwendige Änderungen im Feature-Modell, um diese Fehler zu korrigieren

Diese Fehler können nun dadurch korrigieren werden, dass folgende Constraints zum Feature-Modell hinzugefügt werden. Somit wird sichergestellt, dass für die Zugriffe benötigte Ziele in jedem Produkt vorhanden sind, in dem die zugreifenden Features vorhanden sind.

- Generator ⇒ Solver ∧ Undo ∧ States
- Undo ⇒ States
- Solver ⇒ Undo ∧ States

## 6.10 TankWar

Das Feature-Modell der Softwareproduktlinie **TankWar** ist in Abbildung 12 zu sehen.

TankWar ist ein Echtzeit-Strategie-Kriegsspiel [Kol11].

In der GUIDSL-Datei, die dieses Feature-Modell als Grammatik beschreibt, findet sich als erste Zeile `//NoAbstractFeatures`. Dies ist kein simpler Kommentar, sondern stellt sicher, dass alle Features als konkrete Features behandelt werden.

### Analyse der Fehlerausgabe

An dieser Stelle wird eine Fehlermeldung exemplarisch analysiert:

```
.../TankWar/features/Image/ExplodierenEffekt.java:21:
  Semantic Error: MAYBE dependency:
Feature Image accesses the method
  protected void explodieren() { ... }
of feature explodieren.
Feature explodieren may not be present in every valid selection.
```

Die Klasse `ExplodierenEffekt` des Features IMAGE greift in Zeile 21 auf die Methode `protected void explodieren()` der Klasse `ExplodierenEffekt` des Features EXPLODIEREN zu. Die benötigte Methode `protected void explodieren()` ist innerhalb der Klasse `ExplodierenEffekt` nur im Feature EXPLODIEREN vorhanden. Wenn das Feature IMAGE ausgewählt wurde, ist nicht sichergestellt, dass auch das Feature EXPLODIEREN ausgewählt wurde, da das Feature EXPLODIEREN im Feature-Modell als optional gekennzeichnet ist und kein Constraint im Feature-Modell vorhanden ist, das aufgrund der Auswahl des Features IMAGE die Auswahl des Features EXPLODIEREN mittels einer Implikation IMAGE $\Rightarrow$ EXPLODIEREN erzwingen würde. Diese *maybe*-Beziehung der beiden Features resultiert in einem Typfehler, da es somit laut Feature-Modell valide Produkte geben kann, die einen Kompilerfehler erzeugen, da der Methodenzugriff ins Leere geht, wenn das Zielfeature nicht ausgewählt wurde.

Eine komplette Liste der Fehlermeldungen und eine komplette Analyse der Fehlermeldungen ist auf der beiliegenden CD zu finden.

### Notwendige Änderungen im Feature-Modell, um diese Fehler zu korrigieren

Diese Fehler können nun dadurch korrigieren werden, dass folgende Constraints zum Feature-Modell hinzugefügt werden. Somit wird sichergestellt, dass für die Zugriffe benötigte Ziele in jedem Produkt vorhanden sind, in dem die zugreifenden Features vorhanden sind.

- PC $\Leftrightarrow$ FUER_PC
- HANDY $\Leftrightarrow$ FUER_HANDY
- IMAGE $\Rightarrow$ EXPLODIEREN
- PC $\Leftrightarrow$ RE_FUER_PC

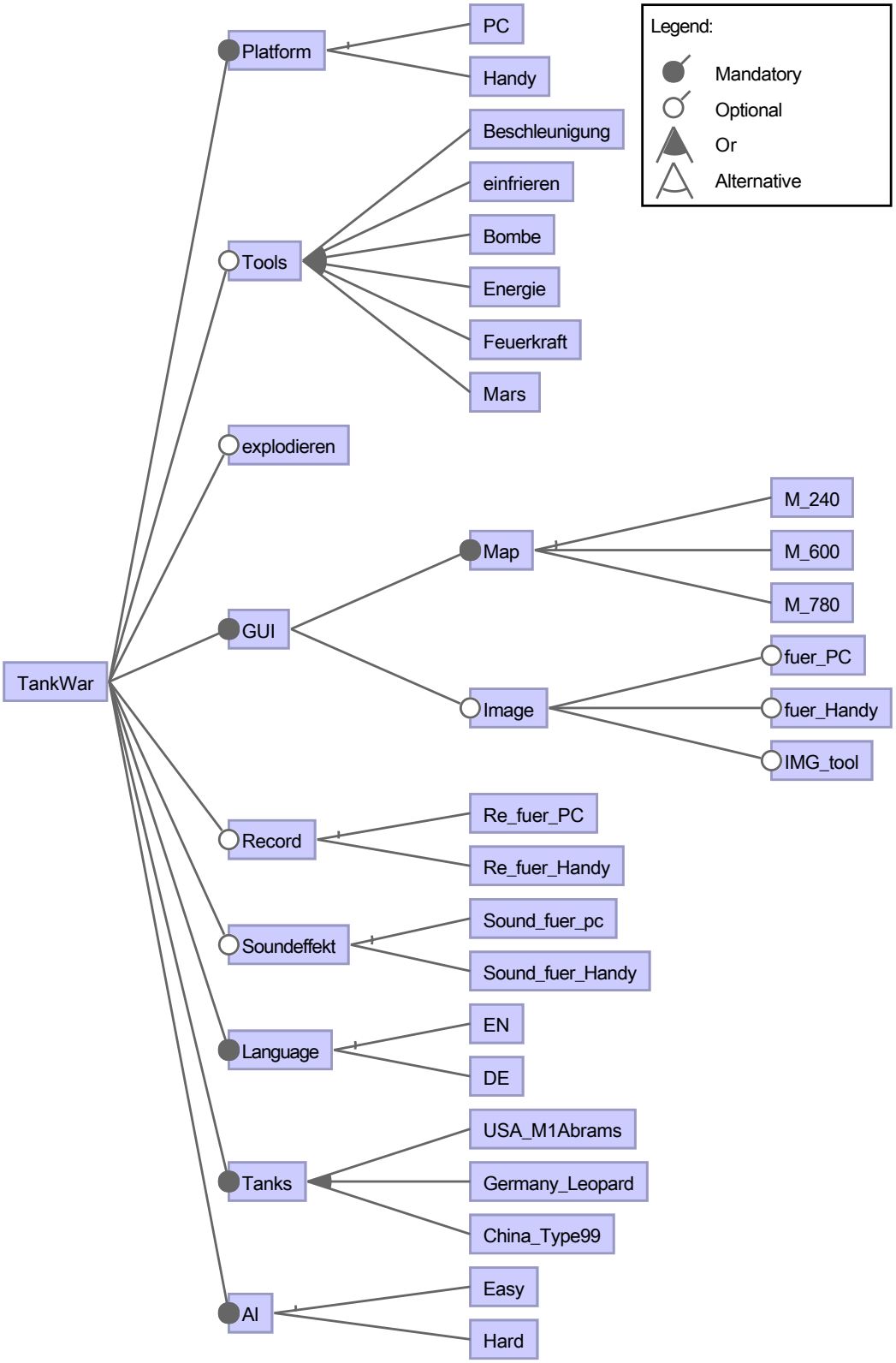



Abbildung 12: TankWar-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13]).
Die Kennzeichnung der Alternativ-Gruppen ist leider nur undeutlich zu sehen (angedeutete vertikale Striche), bedingt durch das gewählte Layout aber nicht besser darstellbar.

- Handy ⇔ Re_fuer_Handy
- IMG_tool ⇒ Tools

### False Positives

Bei der Typüberprüfung fehlten die Bibliotheken, aus denen mittels `import` Klassen verwendet werden. TankWar bindet im Feature Handy eine Klasse namens `MIDlet` ein, stellt eine gleichbenannte Klasse aber auch als leeren Rumpf im alternativen Feature PC zur Verfügung. Die Klasse `MIDlet` steht somit in jedem Produkt zur Verfügung. JastAddJ sieht den leeren Rumpf fälschlicherweise als einziges Ziel an, da die Import-Statements nicht ausgewertet werden (siehe Abschnitt 7.1 auf Seite 62). Aus diesem Umstand resultieren folgende falsche Fehlermeldungen, die auch nach dem Hinzufügen der oben genannten Constraints zum Feature-Modell erhalten bleiben.

```
.../TankWar/features/TankWar/TankManager.java:42:
  Semantic Error: MAYBE dependency:
Feature TankWar accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/Handy/TankManager.java:5:
  Semantic Error: NEVER dependency:
Feature Handy accesses the super constructor
  public MIDlet()
of feature PC.
Features Handy and PC cannot both be present in a valid selection.

.../TankWar/features/Tools/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature Tools accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/Beschleunigung/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature Beschleunigung accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/einfrieren/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature einfrieren accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/Bombe/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature Bombe accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/Energie/TankManager.java:5:
```

```
  Semantic Error: MAYBE dependency:
Feature Energie accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/Feuerkraft/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature Feuerkraft accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/Mars/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature Mars accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/explodieren/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature explodieren accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/Record/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature Record accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/Sound_fuer_pc/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature Sound_fuer_pc accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/Sound_fuer_Handy/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature Sound_fuer_Handy accesses the super constructor
  public MIDlet()
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/TankWar/TankManager.java:5:
  Semantic Error: MAYBE dependency:
Feature TankWar accesses the type
  (default package).MIDlet
of feature PC.
Feature PC may not be present in every valid selection.

.../TankWar/features/Handy/Maler.java:31:
  Semantic Error: NEVER dependency:
Feature Handy accesses the type
  (default package).MIDlet
of feature PC.
Features Handy and PC cannot both be present in a valid selection.
```

## 6.11 Violet

Das Feature-Modell der Softwareproduktlinie **Violet** ist in Abbildung 13 zu sehen.
Violet ist ein einfacher UML-Editor [Kol11].

### Analyse der Fehlerausgabe

An dieser Stelle wird eine Fehlermeldung exemplarisch analysiert:

```
.../Violet/features/HelpMenu/com/horstmann/violet/framework/EditorFrame.java:7,18:
  Semantic Error: MAYBE dependency:
Feature HelpMenu accesses the field
  protected ResourceFactory factory;
of feature MenuResources.
Feature MenuResources may not be present in every valid selection.
```

Die Klasse `com.horstmann.violet.framework.EditorFrame` des Features HelpMenu greift in Zeile 7 auf das Feld `protected ResourceFactory factory;` der Klasse `com.horstmann.violet.framework.EditorFrame` des Features MenuResources zu. Das benötigte Feld `protected ResourceFactory factory;` ist innerhalb der Klasse `com.horstmann.violet.framework.EditorFrame` nur im Feature MenuResources vorhanden. Wenn das Feature HelpMenu ausgewählt wurde, ist nicht sichergestellt, dass auch das Feature MenuResources ausgewählt wurde, da das Feature MenuResources im Feature-Modell als optional gekennzeichnet ist und kein Constraint im Feature-Modell vorhanden ist, das aufgrund der Auswahl des Features HelpMenu die Auswahl des Features MenuResources mittels einer Implikation HelpMenu $\Rightarrow$ MenuResources erzwingen würde. Diese *maybe*-Beziehung der beiden Features resultiert in einem Typfehler, da es somit laut Feature-Modell valide Produkte geben kann, die einen Kompilerfehler erzeugen, da der Feldzugriff ins Leere geht, wenn das Zielfeature nicht ausgewählt wurde.

Eine komplette Liste der Fehlermeldungen und eine komplette Analyse der Fehlermeldungen ist auf der beiliegenden CD zu finden.

### Notwendige Änderungen im Feature-Modell, um diese Fehler zu korrigieren

Diese Fehler können nun dadurch korrigieren werden, dass folgende Constraints zum Feature-Modell hinzugefügt werden. Somit wird sichergestellt, dass für die Zugriffe benötigte Ziele in jedem Produkt vorhanden sind, in dem die zugreifenden Features vorhanden sind.

- HelpMenu $\Rightarrow$ MenuResources
- WindowMenu $\Rightarrow$ MenuResources
- ViewMenu $\Rightarrow$ MenuResources
- EditMenu $\Rightarrow$ MenuResources
- FileMenu $\Rightarrow$ MenuResources

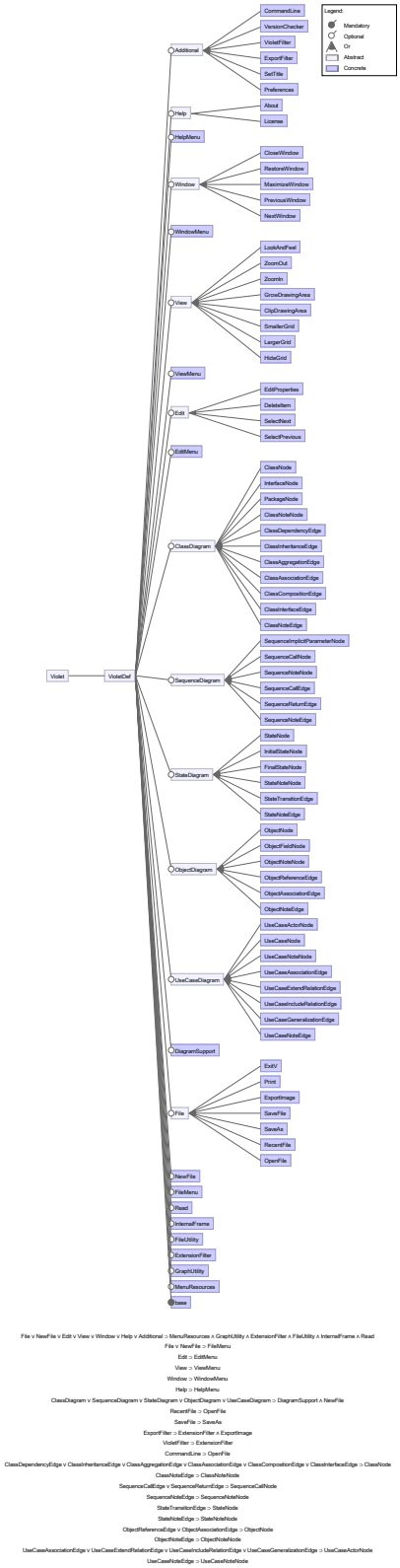



Abbildung 13: Violet-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13])

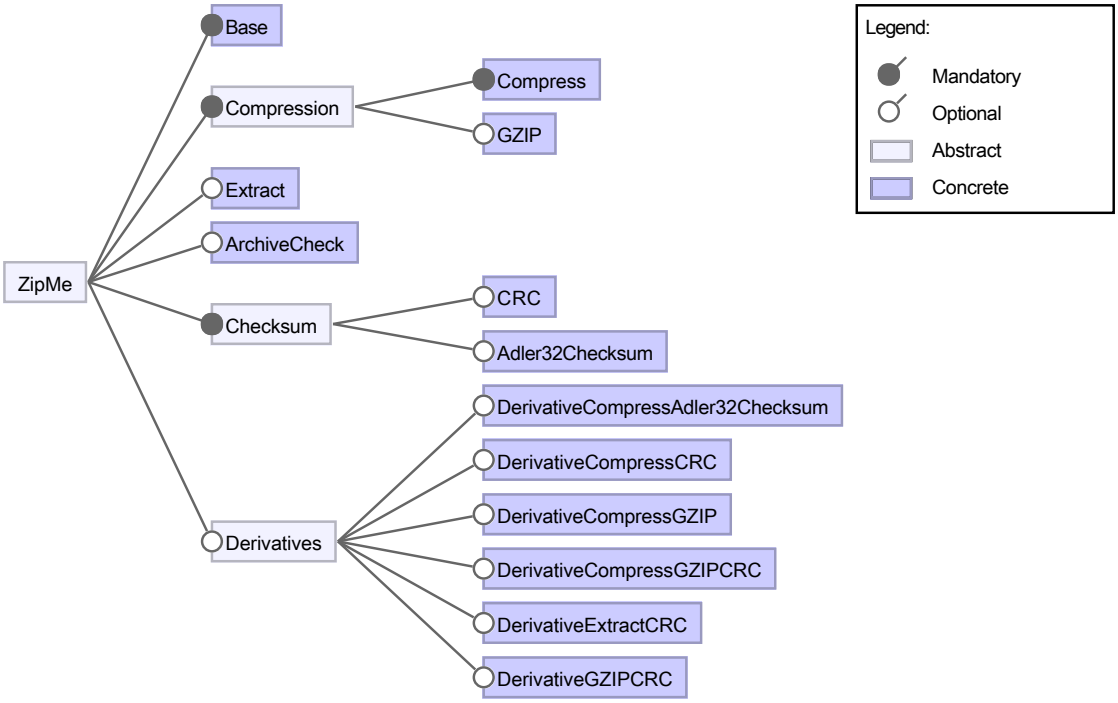
- DiagramSupport ⇒ NewFile ∧ MenuResources ∧ GraphUtility ∧ InternalFrame
- InternalFrame ⇒ SetTitle
- ExtensionFilter ⇒ MenuResources
- Read ⇒ GraphUtility
- FileUtility ⇒ ExtensionFilter
- GraphUtility ⇒ FileUtility

## 6.12 ZipMe

Das Feature-Modell der Softwareproduktlinie **ZipMe** ist in Abbildung 14 zu sehen.

ZipMe ist eine Java-Bibliothek für die Dateikomprimierung in Java-Mobile-Anwendungen [Kol11].

Bei der Typprüfung ergaben sich keine Fehlermeldungen. Auch bei der Komposition und der anschließenden Kompilierung mit Fuji ergaben sich in den einzelnen 24 Varianten keine Fehlermeldungen.




$$GZIP \Rightarrow CRC$$

$$Compress \wedge Adler32Checksum \Leftrightarrow DerivativeCompressAdler32Checksum$$

$$Compress \wedge CRC \Leftrightarrow DerivativeCompressCRC$$

$$Compress \wedge GZIP \Leftrightarrow DerivativeCompressGZIP$$

$$Compress \wedge GZIP \wedge CRC \Leftrightarrow DerivativeCompressGZIPCRC$$

$$Extract \wedge CRC \Leftrightarrow DerivativeExtractCRC$$

$$GZIP \wedge CRC \Leftrightarrow DerivativeGZIPCRC$$

Abbildung 14: ZipMe-Feature-Diagramm (erstellt mit FeatureIDE [Thü+13])

Holthusen [Hol12] meldet zwei Fehler im Zusammenhang mit dem Feature CRC in der Klasse `CRC32`. Diese Fehler wurden anscheinend durch das Constraint $GZIP \Rightarrow CRC$ im Feature-Modell gelöst. Wird dieses Constraint aus dem Feature-Modell herausgenom-

men, dann meldet der Typchecker folgende Fehler:

```
.../GZIP/net/sf/zipme/GZIPInputStream.java:43:
  Semantic Error: MAYBE dependency:
Feature GZIP accesses the type
  net.sf.zipme.CRC32
of feature CRC.
Feature CRC is not present in every valid selection.

.../DerivativeCompressGZIP/net/sf/zipme/GZIPOutputStream.java:18:
  Semantic Error: MAYBE dependency:
Feature DerivativeCompressGZIP accesses the type
  net.sf.zipme.CRC32
of feature CRC.
Feature CRC is not present in every valid selection.
```

## 6.13 Zeitmessungen

Um die Performanz des entwickelten Typcheckers zu ermitteln, wurde die Zeit gemessen, die der Typchecker benötigt, um eine Softwareproduktlinie zu analysieren (familienbasierter Ansatz). Diese Zeit wurde verglichen mit der Gesamtzeit, die es braucht, jedes valide Produkt der Softwareproduktlinie mit Fuji zu komponieren und zu kompilieren (produktbasierter Ansatz). Alle validen Produkte einer Produktlinie wurden zuvor mit FeatureIDE ermittelt. Diese Messung wurde für jede Softwareproduktlinie einmal durchgeführt. Bei mehrmaligen Messungen einer Softwareproduktlinie waren die Zeitabweichungen minimal. Die Messungen wurden für alle zwölf betrachteten Softwareproduktlinien durchgeführt. Da die Anzahl der validen Produkte von Violet nicht ermittelt werden konnte, fand dafür auch keine Messung statt.

### 6.13.1 Änderungen an Main.java für die Zeitmessung

In der Klasse `Main.java` von Fuji wurde die Zeit gemessen, die es braucht, den *Abstract Syntax Tree* aufzubauen, und die Zeit, die benötigt wird, um alle Fehlermeldungen zu erhalten.

Auf der Konsole wurden mittels des Programms `time` die Gesamtzeiten gemessen und aus der Differenz der Summe von User- und System-Zeit und der in der Klasse `Main.java` gemessenen Zeiten die Zeit ermittelt, die es braucht, die JVM zu starten.

### 6.13.2 Verwendeter Rechner für die Zeitmessung

Die Zeitmessungen wurden durchgeführt auf einem AMD Athlon 64 Processor 3200+, 2.2 GHz, 2GB RAM.

### 6.13.3 Messung

Bei den Softwareproduktlinien EPL, GPL, Notepad, Prevayler und Violet musste der Inhalt der Config-Dateien, d.h. der Textdateien, die alle zu komponierenden Features eines Produkts auflisten, umgedreht werden, so dass sichergestellt war, dass die Komposition mit dem Basis-Feature beginnt. In den anderen Softwareproduktlinien war dies bereits gegeben.

### 6.13.4 Ergebnisse

Siehe die Tabelle 3 auf Seite 60.

Tabelle 3: Ergebnisse der Benchmark-Messungen

| | | | | familienbasiert | | | | produktbasiert[1] | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SPL | Anzahl Features | Lines of Code | Anzahl Produkte | construct AST | splError-Check | startJvm | $\sum^2$ | construct AST | errorCheck | startJvm | $\sum^3$ |
| EPL | 12 | 111 | 425 | 190 ms | 12180 ms | 9711 ms | < 1 min | 22860 ms | 303440 ms | 285370 ms | 10 min |
| GPL | 27 | 1930 | 156 | 270 ms | 1823380 ms | 476186 ms | 38 min | 10210 ms | 189120 ms | 245262 ms | 7 min |
| Graph | 5 | 251 | 16 | 140 ms | 3080 ms | 3432 ms | < 1 min | 790 ms | 13990 ms | 16286 ms | < 1 min |
| GUIDSL | 26 | 10084 | 24 | 260 ms | 4154960 ms | 690382 ms | 81 min | 2450 ms | 90540 ms | 135000 ms | 4 min |
| Notepad | 12 | 937 | 512 | 200 ms | 81010 ms | 36697 ms | 2 min | 29760 ms | 903860 ms | 1398962 ms | 39 min |
| PKJab | 8 | 3373 | 48 | 170 ms | 112170 ms | 53434 ms | 3 min | 3930 ms | 137100 ms | 220465 ms | 6 min |
| Prevayler | 6 | 5268 | 32 | 170 ms | 52630 ms | 31313 ms | 1 min | 2950 ms | 108510 ms | 180363 ms | 5 min |
| Raroscope | 5 | 316 | 16 | 140 ms | 2620 ms | 3292 ms | < 1 min | 900 ms | 15840 ms | 20534 ms | < 1 min |
| Sudoku | 7 | 1422 | 64 | 130 ms | 20240 ms | 17721 ms | < 1 min | 3730 ms | 126070 ms | 199145 ms | 5 min |
| TankWar | 37 | 4845 | 2458 | 160 ms | 6258690 ms | 1274416 ms | 126 min | 148630 ms | 4925820 ms | 6806325 ms | 198 min |
| Violet | 88 | 7151 | — | 360 ms | 117034050 ms | 11106127 ms | 2136 min | — | — | — | — |
| ZipMe | 13 | 3446 | 24 | 220 ms | 379940 ms | 123795 ms | 8 min | 1660 ms | 47040 ms | 61811 ms | 2 min |

[1] Werte aller Produkte zusammenaddiert
[2] Zeiten für familienbasierten Ansatz zusammen; gerundet
[3] Zeiten für produktbasierten Ansatz (alle Produkte) zusammen; gerundet

### 6.13.5 Zusammenfassung

Bei kleinen und mittelgroßen Softwareproduktlinien mit wenigen Produkten (ZipMe, GPL, GUIDSL) ist die Zeit für die Ausführung des familienbasierten Typcheckers deutlich länger als die Erzeugung aller Varianten (vor allem bei großen, komplizierten Feature-Modellen). GUIDSL hat wenige Produkte und ein Feature-Modell, das durch die vielen Constraints lange Laufzeiten des SAT-Solvers hervorruft.

Bei den Fehlermeldungen der einzelnen Produkte ist es nicht einfach, die Fehler herauszuarbeiten, um vor allem zu erkennen, welches Feature welche anderen Features braucht. Es ist anhand dieser Fehlermeldungen nicht erkennbar, in welchem Feature sich ein benötigter Typ, ein benötigtes Feld oder eine benötigte Methode befindet. Der familienbasierte Typchecker liefert hierbei deutlich bessere, aussagekräftigere Informationen, die deutlich machen, worin das Problem besteht. Somit müssen die Zusammenhänge nicht erst umständlich erarbeitet werden.

Bei Softwareproduktlinien mit vielen Produkten ist der familienbasierte Typchecker um den Faktor 10 bis 20 schneller (EPL, Notepad). Wenn sich wie bei Violet die Anzahl der Produkte nicht ermitteln lässt und damit die produktbasierte Typprüfung mit allen Produkten nicht durchgeführt wird, liefert der familienbasierte Typchecker trotz langer Laufzeit eine brauchbare (und möglicherweise die einzig durchführbare) Möglichkeit zur Typprüfung.

# 7 Offene Probleme

## 7.1 *False Positives*

- Wenn in einem Interface keinerlei Felder oder Methoden deklariert werden, so wird dieser Interface-Rumpf durch den Typchecker nicht als Ziel für einen Typzugriff erkannt. Die Ermittlung der Features, die eine bestimmte Klasse oder ein bestimmtes Interface implementieren, erfolgt anhand der Felder oder Methoden, die in der komponierten Klasse oder im komponierten Interface vorhanden sind. Dieses Problem tritt für Klassen nicht auf, da in einer scheinbar leeren Klasse implizit ein Default-Konstruktor vorhanden ist. Dieses Problem führt zu falschen Fehlermeldungen in der Softwareproduktlinie EPL (siehe Abschnitt 6.1 auf Seite 36). Möglicherweise lässt sich dieses Problem lösen, indem die durch Fuji gesammelten Informationen über Rollen genutzt werden.

- Import-Statements werden in der vorliegenden Implementierung des Typcheckers nicht ausgewertet, da nicht ermittelt werden kann, welches Feature ein bestimmtes Import-Statement in die Klasse oder das Interface einführt. Ein Import-Statement im *Abstract Syntax Tree* wird durch Fuji dem Feature zugerechnet, das als erstes zu einer Klasse oder einem Interface beiträgt. Dadurch ist es nicht möglich, herauszufinden, ob in einem konkreten Feature benötigte Imports von Klassen oder Interfaces vorhanden sind. Dieses Problem trägt zu den falschen Fehlermeldungen der Softwareproduktlinie TankWar bei (siehe Abschnitt 6.10 auf Seite 52).

- Probleme mit Methoden, die in Java-Standardklassen vorhanden sind und in einem Feature überschrieben werden, werden im Abschnitt 6.1 auf Seite 37 angesprochen.

- Probleme mit alternativen Typhierarchien durch verschiedene Subtypinformationen in alternativen Features werden im Abschnitt 5.1.3 auf Seite 28 angesprochen.

## 7.2 Verbesserung der Fehlerausgabe

Im Typchecker wurde die Methode `addSplErrors` vor allem im Hinblick auf die Ausgabe der Fehlermeldungen möglichst generisch gehalten, um damit viele Fehlermeldungen abdecken zu können. Darunter leidet allerdings die Ausdruckskraft der Fehlerausgabe. Es wäre möglich, die Fehlerausgabe detaillierter und spezifischer für die einzelnen Fehlermeldungen zu gestalten, etwa indem ein spezieller Fehlercode an die Methode `addSplErrors` übergeben und in der Methode ausgewertet wird. Allerdings ginge dadurch der gegenwärtige Vorteil verloren, dass neue Arten von Zugriffen einfach hinzugefügt werden können, ohne dass die Methode `addSplErrors` angepasst werden muss. Bei der Verwendung von Fehlercodes müsste für neue Fehlercodes die Methode `addSplErrors` angepasst werden. Als Kompromiss könnte ein spezieller Fehlercode vorgehalten werden, der eine generische Fehlermeldung erzeugt.

Um die Ausgabe der Fehlermeldungen ausführlicher zu gestalten, könnte für Fehlermeldungen, die sich auf Probleme mit Subtypinformationen beziehen, noch folgende

Zeile hinzugefügt werden: „The information that class $x$ extends class/implements interface/extends interface $y$ **is missing** in feature $z$.“ Damit würde verdeutlicht, in welchem Feature die Information fehlt, anstatt nur zu sagen, wo die Information ausschließlich vorhanden ist. Alternativ könnte auch zur bisherigen Ausgabe folgendes angefügt werden: „... is only present in feature $x$ **therefore missing in feature** $y$“.

## 7.3 Verbesserungen der Typchecks

*Try-Catch*-Blöcke könnten daraufhin untersucht werden, ob die Information, dass geworfene Exceptions Subtypen der gefangenen Exceptions sind, in allen Produkten vorhanden ist, in denen diese Information benötigt wird.

## 7.4 Performanz-Verbesserungen

### 7.4.1 Cachen von Anfragen an den SAT-Solver

Mitunter kann es zu vielen identischen Anfragen an den SAT-Solver kommen, etwa, wenn in einer Methode eine andere Methode in einem anderen Feature oft hintereinander aufgerufen wird.

Beispielsweise wird in der Softwareproduktlinie TankWar im Feature IMAGE in der Klasse `Tank.java` in der Methode `tankMalen()` die Methode `drawImage` mehrmals in einer Fallunterscheidung aufgerufen.

Durch ein Cachen von identischen Anfragen an den SAT-Solver könnte die Performanz des Typcheckers in solchen Fällen deutlich gesteigert werden.

### 7.4.2 Ermittlung der Pfade in der Typ-Hierarchie nur einmal für jedes Paar aus Typen

In einer Softwareproduktlinie könnten die Pfade in der Typ-Hierarchie für jedes Paar aus Typen beim ersten Aufruf der Methode `getTypeHierarchyPaths` global gespeichert werden, um diese Information nicht bei jedem Aufruf der Methode `getTypeHierarchyPaths` neu berechnen zu müssen.

### 7.4.3 Mehr Tests bei der Ermittlung der Zugriffe durchführen

Anstatt bestimmte Tests in der Methode `addSplErrors` durchzuführen, könnten diese Tests bereits bei der Ermittlung der Zugriffe in der entsprechenden `splTypeCheck`-Methode durchgeführt werden. Findet beispielsweise die Überprüfung, ob es sich bei dem Quellfeature und dem Zielfeature des Zugriffs um dasselbe Feature handelt, bereits in der Methode `splTypeCheck` statt, so müsste für den positiven Fall die Methode `addSplErrors` nicht mehr aufgerufen werden (da sich dadurch kein Typfehler ergeben kann). Diese Vorgehensweise würde aber die `splTypeCheck`-Methoden verkomplizieren und in dupliziertem Code für jede dieser Methoden resultieren. Letztendlich ist hier eine Abwägung von Wartbarkeit gegenüber einer Performanzverbesserung nötig.

### 7.4.4 Abtrennung des zweiten Schritts vom ersten

Durch eine Abtrennung des zweiten Schritts der Typprüfung (um Probleme mit Subtypinformationen zu analysieren) vom ersten Schritt könnte die Typprüfung auch ohne den zweiten Schritt durchgeführt werden. Da der zweite Schritt die Durchführung der Typprüfung deutlich verlängert und die dabei gefundenen Typfehler möglicherweise durch Korrektur der Typfehler aus dem ersten Schritt bereits beseitigt werden (etwa indem nötige Constraints zum Feature-Modell hinzugefügt werden), wäre damit eine gestaffelte Typprüfung möglich. Der zweite Schritt, der dann seltener ausgeführt würde, könnte durch eine neue Kommandozeilen-Option beim Aufruf des Typcheckers zugeschaltet werden.

## 7.5 Architektur-Entscheidungen

### 7.5.1 Aufteilung des Aspekts `ExtendedTypeCheck`

Der Aspekt `ExtendedTypeCheck` könnte auf mehrere Aspekte aufgeteilt werden, um die Erweiterbarkeit des Typcheckers um neue Analysen von Zugriffen zu betonen. Dazu könnten alle `splErrorChecks` in einen Aspekt (oder mehrere Aspekte) gekapselt werden, die Methode `addSplErrors` und weiterführende Methoden in einem anderen Aspekt gekapselt werden und alle anderen Methoden in einem dritten Aspekt gekapselt werden.

## 7.6 Automatische Schlussfolgerungen aus den Fehlermeldungen

Für einfache Fälle können aus den Fehlermeldungen automatisch Constraints abgeleitet werden, die – zum Feature-Modell hinzugefügt – die Typfehler korrigieren. Wenn beispielsweise ein Feature auf ein optionales Feature zugreift, kann der Typfehler durch eine Implikation korrigiert werden. Der Typchecker könnte so für Softwareproduktlinien, die nur solche einfachen Typfehler enthalten, ein korrigiertes Feature-Modell vorschlagen.

## 7.7 Offene TODOs

- Bei der aktuellen Implementierung muss noch beachtet werden, dass der Typchecker mit der Option `-typechecker` *zusammen* mit der Option `-fopRefs` aufgerufen werden muss. Dies könnte man dahingehend ändern, dass die Option `-typechecker` die Option `-fopRefs` im Quelltext der Datei `Main.java` impliziert, so dass die Option `-fopRefs` nicht mehr explizit ausgewählt werden muss.

- Wenn die Softwareproduktlinie Typfehler enthält, die auch ohne Bezug zum Feature-Modell Kompilerfehler erzeugen, dann bricht der Typchecker bereits in einer frühen Phase in Fuji ohne Fehlermeldungsausgabe ab. Deshalb muss zunächst sichergestellt werden, dass die Softwareproduktlinie frei von diesen allgemeinen Kompilerfehlern ist.

# 8 Verwandte wissenschaftliche Arbeiten

Sergiy Kolesnikov, [Kol11], stellt den erweiterbaren Java-Compiler Fuji für feature-orientierte Softwareproduktlinien vor, auf dem der in dieser Arbeit entwickelte Typchecker aufbaut. Dieser Typchecker stellt damit eine Erweiterung für Fuji dar, der Gebrauch macht von einer Erweiterung von Fuji, mit der Referenzen zwischen den Knoten des *Abstract Syntax Tree*s von JastAddJ ermittelt werden.

Sönke Holthusen, [Hol12] beschreibt einen familienbasierten Typchecker für die Eclipse-Erweiterung FeatureIDE, mit der Software-Produktlinien modelliert und implementiert werden können. Der Typchecker wurde in Form eines Plugins für FeatureIDE entwickelt.

Es existieren bereits einige formale Ansätze für Typsysteme in feature-orientierten Softwareproduktlinien. Diese beziehen sich aber nur auf funktionsreduzierte Untermengen von Programmierssprachen. So definiert [Ape+10b] beispielsweise ein Typsystem für feature-orientierte Produktlinien für eine Untermenge von Java, das $\text{FFJ}_{PL}$ genannt wird.

# 9 Qualitätssicherung

Für jeden Typfehler, den der Typchecker finden soll, wurde ein Testfall in Form einer kleinen Softwareproduktlinie geschrieben, die ein Minimalbeispiel für einen solchen Typfehler darstellt. Diese 29 Testfälle sind im Anhang auf der beiliegenden CD beschrieben.

Diese Testfälle dienen als Regressionstests. Die Fehlermeldungen, die der Typchecker für jeden dieser Testfälle finden soll, sind jeweils in der Datei `expectedErrors.txt` aufgelistet. Nach jeder Änderung am Typchecker können diese Testfälle durchlaufen werden, um sicherzustellen, dass immer noch die gleichen Fehlermeldungen erzeugt werden. Mittels eines Aufrufs eines Shell-Skripts können diese Testfälle automatisch durchlaufen werden. Das Skript vergleicht die ausgegebenen Fehlermeldungen des Typcheckers mit den erwarteten Fehlermeldungen. Da sich die Reihenfolge der Fehlermeldungen ändern kann, erfolgt der Vergleich bei Nichtübereinstimmung nochmals in sortierter Form der Ausgabe und der erwarteten Fehlermeldungen. Die unterschiedliche Reihenfolge ergibt sich aufgrund der Verwendung von `HashMaps` im Typchecker, die ihre Elemente in einer *for-each*-Schleife bei mehrmaligen Durchläufen in keiner festen Reihenfolge zurückgeben.

# 10 Zusammenfassung

In dieser Arbeit wurde ein Typchecker für Fuji vorgestellt, der anhand des *Abstract Syntax Tree* einer Softwareproduktlinie Zugriffe zwischen Features analysiert und mit den Abhängigkeiten im Feature-Modell abgleicht.

Dabei werden Typfehler entdeckt, wenn Features auf andere Features zugreifen, die aufgrund der Abhängigkeiten im Feature-Modell nicht für jedes Produkt ausgewählt werden müssen, in denen sie benötigt werden. Diese Typfehler werden gemeldet. Die Fehlermeldungen liefern der Erstellerin der Softwareproduktlinie genügend Informationen, um diese Fehler zu erkennen und zu korrigieren. Dies kann beispielsweise durch einfaches Hinzufügen von weiteren Abhängigkeiten zwischen Features zum Feature-Modell erfolgen.

Der Typchecker wurde anhand von zwölf vorhandenen Softwareproduktlinien unterschiedlicher Größe und unterschiedlicher Anzahl an Features getestet. Dabei fanden sich in kleinen Softwareproduktlinien kaum Typfehler (mit Ausnahme von EPL). Bei größeren Softwareproduktlinien stieg die Anzahl der gefundenen Typfehler mit der Größe und der Anzahl der Features. Auffallend war dabei, dass Features, die notwendigen Quelltext für Kombinationen von Features enthalten (sogenannte *Derivatives*), bei Auswahl einer solchen Kombination aufgrund von Abhängigkeiten im Feature-Modell zwingend ausgewählt werden müssen. Die Abhängigkeiten, die den umgekehrten Fall beschreiben, sind aber im Feature-Modell meist nicht berücksichtigt. Dadurch kommt es zu Typfehlern bei der Auswahl von Features mit Kombinations-Quelltext, wenn die benötigten Features, die Teil der Kombination sind, nicht ausgewählt werden müssen.

Der familienbasierte Ansatz dieses Typcheckers wurde anschließend mit dem produktbasierten Ansatz, bei dem alle validen Produkte komponiert und kompiliert werden, verglichen. Beim Vergleich der Zeiten dieser Ansätze zeigte sich, dass der familienbasierte Typchecker ab einer gewissen Größe der Softwareproduktlinie und einer gewissen Anzahl an Produkten dem produktbasierten Ansatz deutlich überlegen ist. Dies war insbesondere dann auffällig, wenn die Anzahl an validen Produkten einer Softwareproduktlinie aufgrund der hohen Anzahl an optionalen Features nicht mehr ohne Weiteres bestimmt werden konnte.

Abschließend wurden Ideen für eine Verbesserung und Erweiterung der bestehenden Implementierung aufgezeigt. Insbesondere im Caching von Anfragen an den SAT-Solver dürfte reichlich Potential für Performanzverbesserungen liegen.

Der Typchecker ist aufgrund seiner modularen Architektur, die von JastAddJ und Fuji übernommen wurde, leicht erweiterbar.

## Literaturverzeichnis


[AK09] Sven Apel und Christian Kästner. „An Overview of Feature-Oriented Software Development“. In: *Journal of Object Technology (JOT)* 8.5 (2009), S. 49–84 (siehe S. 6, 9, 10, 12).

[AKS11] Sven Apel, Christian Kästner und Gunter Saake. *Software Productline Engineering*. Vorlesung, Sommersemester 2011, Universität Passau. 2011 (siehe S. 11).

[Aho+07] Alfred V Aho, Monica S Lam, Ravi Sethi und Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Bd. 1009. Pearson/Addison Wesley, 2007 (siehe S. 12).

[Ape+10a] Sven Apel, Wolfgang Scholz, Christian Lengauer und Christian Kästner. „Language-independent reference checking in software product lines“. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. ACM. 2010, S. 65–71 (siehe S. 14).

[Ape+10b] Sven Apel, Christian Kästner, Armin Größlinger und Christian Lengauer. „Type safety for feature-oriented product lines“. English. In: *Automated Software Engineering* 17.3 (2010), S. 251–300. ISSN: 0928-8910. DOI: 10.1007/s10515-010-0066-8. URL: http://dx.doi.org/10.1007/s10515-010-0066-8 (siehe S. 11, 65).

[Bat05] Don Batory. „Feature Models, Grammars, and Propositional Formulas“. In: *Software Product Lines*. Hrsg. von Henk Obbink und Klaus Pohl. Bd. 3714. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 7–20. ISBN: 978-3-540-28936-4. DOI: 10.1007/11554844_3. URL: http://dx.doi.org/10.1007/11554844_3 (siehe S. 9, 10).

[CN02] P. Clements und L.M. Northrop. *Software Product Lines: Practices and Patterns*. The SEI Series in Software Engineering. Addison-Wesley, 2002. ISBN: 9780201703320 (siehe S. 6, 9).

[DB07] Mark Dalgarno und Danilo Beuche. „Variant Management“. In: *The British Computer Society. 3rd British Computer Society Configuration Management Specialist Group Conference. Oxford*. 2007 (siehe S. 6).

[EH07] Torbjörn Ekman und Görel Hedin. „The jastadd extensible java compiler“. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. OOPSLA '07. New York, NY, USA: ACM, 2007, S. 1–18. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297029. URL: http://doi.acm.org/10.1145/1297027.1297029 (siehe S. 12).

[Fuj] *Fuji: An Extensible Compiler for Feature-Oriented Programming in Java*. URL: http://fosd.de/fuji (besucht am 10. 03. 2013) (siehe S. 13).

[Hed00] Görel Hedin. „Reference attributed grammars“. In: *Informatica (Slovenia)* 24.3 (2000), S. 301–317 (siehe S. 12).

[Hol12] Sönke Holthusen. „Typsicherheit in Feature-orientierten Software-Produktlinien in FeatureIDE“. Masterarbeit. Otto-von-Guericke-Universität Magdeburg, Juli 2012 (siehe S. 10, 12, 19, 57, 65).

[JF88] Ralph E Johnson und Brian Foote. „Designing reusable classes“. In: *Journal of object-oriented programming* 1.2 (1988), S. 22–35 (siehe S. 10).

[Jas] *JastAddJ: The JastAdd Extensible Java Compiler*. URL: `http://jastadd.org/web/jastaddj/` (besucht am 10. 03. 2013) (siehe S. 12, 13, 21).

[Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier und John Irwin. *Aspect-oriented programming*. Springer, 1997 (siehe S. 10).

[Kol11] Sergiy Kolesnikov. „An Extensible Compiler for Feature-Oriented Programming in Java“. Masterarbeit. Universität Passau, Feb. 2011 (siehe S. 12, 13, 19, 32, 41, 43–45, 50, 54, 57, 65).

[LHB01] Roberto E Lopez-Herrejon und Don Batory. „A standard problem for evaluating product-line methodologies“. In: *Generative and Component-Based Software Engineering*. Springer, 2001, S. 10–24 (siehe S. 38).

[LHB04] Roberto E Lopez-Herrejon und Don Batory. *The expression problem as a product line and its implementation in AHEAD*. Computer Science Department, University of Texas at Austin, 2004 (siehe S. 14, 35).

[PBL05] Klaus Pohl, Günter Böckle und Frank J van der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer, 2005 (siehe S. 6).

[Pie02] Benjamin C Pierce. *Types and programming languages*. The MIT Press, 2002 (siehe S. 7, 12).

[Pre97] Christian Prehofer. „Feature-oriented programming: A fresh look at objects“. In: *ECOOP'97—Object-Oriented Programming*. Springer, 1997, S. 419–443 (siehe S. 10).

[Rhe+13] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm und Ina Schaefer. „The PLA model: on the combination of product-line analyses“. In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '13. New York, NY, USA: ACM, 2013, 14:1–14:8. ISBN: 978-1-4503-1541-8. DOI: `10.1145/2430502.2430522`. URL: `http://doi.acm.org/10.1145/2430502.2430522` (siehe S. 6).

[SGM02] Clemens Szyperski, Dominik Gruntz und Stephan Murer. *Component software: beyond object-oriented programming*. Addison-Wesley, 2002 (siehe S. 10).

[Sch+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani und Nico Tanzarella. „Delta-oriented programming of software product lines“. In: *Software Product Lines: Going Beyond*. Springer, 2010, S. 77–91 (siehe S. 10).

[Thü+11] Thomas Thüm, Christian Kästner, Sebastian Erdweg und Norbert Siegmund. „Abstract features in feature modeling“. In: *Software Product Line Conference (SPLC), 2011 15th International*. IEEE. 2011, S. 191–200 (siehe S. 9).

[Thü+12] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer und G. Saake. *Analysis strategies for software product lines*. Technical Report FIN-004-2012. University of Magdeburg, 2012 (siehe S. 6, 10, 14).

[Thü+13] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake und Thomas Leich. „FeatureIDE: An Extensible Framework for Feature-Oriented Software Development“. In: *Science of Computer Programming* (2013). To appear; accepted 2012-06-07. (siehe S. 13, 15, 18, 39, 40, 42–45, 47, 48, 51, 55, 57).

# Abbildungsverzeichnis

# Tabellenverzeichnis

## Ordnerstruktur der beiliegenden CD

- Arbeit (Die Ressourcen der Ausarbeitung der Masterarbeit)
- Der ausgelagerte Anhang der Masterarbeit:
  - A. Komplette Fehlerausgabe
  - B. Komplette Analyse der Fehlerausgabe
  - C. Komplette Beschreibung der Testfälle
- Implementierung
  - Dateien des Typcheckers
  - TestCases
  - Benchmark
  - Skripte

## Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. Ich versichere hiermit außerdem, dass diese Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, den 30. April 2013 Peter Lutz