Technische Universität Braunschweig

ISF

Masterarbeit

Metaanalyse zu Strategien für die formale Verifikation von Software-Produktlinien

Vasco Menzel

15. Januar 2016

Institut für Softwaretechnik und Fahrzeuginformatik
Prof. Dr.-Ing. Ina Schaefer

Betreuer:
Dr. Thomas Thüm

Eigenständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Masterarbeit “Metaanalyse zu Strategien für die formale Verifikation von Software-Produktlinien“ selbstständig verfasst sowie alle benutzten Quellen und Hilfsmittel vollständig angegeben habe und dass die Arbeit nicht bereits als Prüfungsarbeit vorgelegen hat.

Braunschweig, den 15. Januar 2016

Kurzfassung

Software-Produktlinien stellen heutzutage ein großes Forschungsgebiet im Bereich der Softwaretechnik dar. Eine Software-Produktlinie (kurz: SPL) dient zur Entwicklung einer Menge ähnlicher Produkte in einer vorher geplanten Vorgehensweise, anstatt jede einzelne Variante von neu auf zu entwickeln. Diese Produkte unterscheiden sich teilweise in ihrer Funktionalität und somit den kundensichtbaren Eigenschaften. Diese Unterschiede werden als Variabilität bezeichnet. Da Software-Produktlinien in allen Anwendungsbereichen verwendet werden, z.B. auch in sicherheitskritischen Bereichen, muss eine Erzeugung zuverlässiger Produkte garantiert werden. Dabei erzeugt die Variabilität zusätzliche Komplexität bei der Verifikation. Daher wurden in den letzten Jahren viele Ansätze veröffentlicht, in denen statische Analysetechniken aus der ursprünglichen Software-Entwicklung auf Software-Produktlinien adaptiert wurden. In dieser Arbeit werden die Analysetechniken *Model Checking*, *Type Checking*, *Static Analysis* und *Theorem Proving* für Software-Produktlinien betrachtet.

Die genannten Analysetechniken können mittels verschiedener Strategien für Software-Produktlinien verwendet werden. Beispielsweise können sie auf alle Produkte einer Software-Produktlinie oder isoliert auf jedes Feature einzeln angewendet werden. Diese Analyseansätze wurden vielfach in einzelnen Evaluierungen untersucht. Dabei wurden unterschiedliche Software-Produktlinien verwendet, die sich unter anderem in der Anzahl der Features und der Anzahl der Produkte unterscheiden. Die Evaluierungen resultieren außerdem in unterschiedlichen Ergebnissen wie Speicherverbrauch oder Analysezeit.

Diese Unterschiede in den Evaluierungen erschweren die Vergleichbarkeit. Dadurch wird die Auswahl einer spezifischen Analysestrategie für eine spezifische Software-Produktlinie erschwert. Um diese Hürde zu überwinden, werden in dieser Arbeit die Evaluierungen dieser Analysetechniken untersucht und in Beziehung gesetzt. Ein weiteres Ziel dieser Arbeit ist die Ermittlung, welche Strategie für welche Software-Produktlinien signifikant besser oder schlechter ist. Zum Schluss werden Richtlinien für zukünftige Evaluierungen ermittelt, die zur besseren Vergleichbarkeit dieser Analysemethoden eingehalten werden sollten.

Abstract

Nowadays software product lines represent a major research field in the area of software engineering. A software product line (short: SPL) is used to develop similar products in a preplanned way instead of developing each product from scratch. These products differ partially in their provided functionality. These differences are called variability. Software product lines are used in several areas like safety critical sectors where the development of a correct software system has to be guaranteed. Therefore the variability produces additional complexity for the verification. To adress this, a lot of approaches were proposed in which static analysis techniques were adapted from traditional software engineering to software product lines. In the following work the analysis techniques *Model Checking, Type Checking, Static Analysis* and *Theorem Proving* are considered.

These analysis techniques could be used for software product lines with different strategies. For instance, they could be applied to all products of the software product line or to each single feature in isolation. The analysis approaches were part of many single evaluations. Different software product lines were used in these evaluations which differ in e.g. the number of features or the number of products. Furthermore the evaluations result in different outcomes like memory consumption or analysis time.

These differences hinder a comparation of the evaluations and to put the strategies in relation. This fact hampers the selection of a specific analysis strategy for a specific software product line. In this thesis, the evaluations of this analysis strategies are examined and put in relation to overcome this hurdle. Another goal of this thesis is the investigation which strategy is significant better or worse for which software product line. The thesis concludes with some proposals of guidelines for future evaluations, which should be followed for a better comparison of future analysis methods.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Gliederung 2

2. Grundlagen **3**
2.1. Generierung von Software-Varianten 3
2.1.1. Software-Produktlinien 3
2.1.2. Implementierung der Variabilität 6
2.2. Statische Verifikation 14
2.2.1. Model Checking 14
2.2.2. Type Checking 20
2.2.3. Theorem Proving 22
2.2.4. Static Analysis 24

3. Verifikation von Software-Produktlinien **27**
3.1. Verifikationsstrategien für Software-Produktlinien 28
3.1.1. Produkt-basierte Strategie 28
3.1.2. Familien-basierte Strategie 31
3.1.3. Feature-basierte Strategie 33
3.1.4. Kombinationen der primären Verifikationsstrategien 36
3.1.5. Eigenschaften und Vor- & Nachteile der Strategien 36
3.2. Detaillierte Problemstellung 38
3.3. Untersuchte Dokumente 40
3.3.1. Model Checking 42
3.3.2. Type Checking 45
3.3.3. Static Analysis 46
3.3.4. Theorem Proving 47
3.4. Hypothesen 48
3.4.1. Hypothesen der untersuchten Paper 49
3.4.2. Ermittelte Hypothesen 50
3.5. Zusammenfassung 51

4. Vergleich der Verifikations -strategien **53**
4.1. Untersuchungsvorgang & Methoden zur Hypothesenuntersuchung 53
4.1.1. Wilcoxon-Vorzeichen-Rang-Test 54
4.1.2. Vote Counting 55
4.1.3. Umformung der Hypothesen 56

4.1.4. Effektgrößen . . . 57
4.1.5. Beschreibung des Untersuchungsvorgangs . . . 58
4.2. Untersuchung der Hypothesen bezüglich Model-Checking . . . 60
4.2.1. Zusammenfassung der Primärstudien . . . 60
4.2.2. Allgemeine Meta-Diagramme . . . 61
4.2.3. Vote Counting der Primärstudien . . . 62
4.2.4. Cohens' d zum Vergleich der Primärstudien . . . 66
4.2.5. Untersuchte Hypothesen . . . 68
4.2.6. Meta-Diagramme der Hypothesen . . . 69
4.2.7. Interpretation der Diagramme und Ergebnisse des Signifikanztests . . . 71
4.2.8. Untersuchung der nicht bestätigten Hypothesen . . . 73
4.2.9. Untersuchungsergebnisse bezüglich der Behauptungen der Primärstudien . 74
4.3. Untersuchung der Hypothesen bezüglich Type Checking . . . 74
4.3.1. Zusammenfassung der Primärstudien . . . 75
4.3.2. Allgemeine Meta-Diagramme . . . 75
4.3.3. Vote Counting der Primärstudien . . . 77
4.3.4. Cohens' d zum Vergleich der Primärstudien . . . 78
4.3.5. Untersuchte Hypothesen . . . 79
4.3.6. Meta-Diagramme der Hypothesen . . . 80
4.3.7. Interpretation der Diagramme und Ergebnisse des Signifikanztests . . . 81
4.3.8. Untersuchung der nicht bestätigten Hypothesen . . . 83
4.3.9. Untersuchungsergebnisse bezüglich der Behauptungen der Primärstudien . 84
4.4. Untersuchung der Hypothesen bezüglich Static Analysis . . . 85
4.5. Untersuchung der Hypothesen bezüglich Theorem Proving . . . 87
4.6. Gefahren für die Validität . . . 90
4.6.1. Gefahren für die Validität der Model-Checking-Primärstudien . . . 90
4.6.2. Gefahren für die Validität der Type-Checking-Primärstudien . . . 91
4.6.3. Gefahren für die Validität der Hypothesenuntersuchungen . . . 91
4.7. Zusammenfassung der Untersuchungsergebnisse . . . 96

5. Empfehlungen für zukünftige Evaluierungen 99

6. Verwandte Arbeiten 103

7. Zusammenfassung, Fazit und Ausblick 105

A. Anhang A 109
A.1. Kurzbeschreibung der Ansätze . . . 109
A.1.1. Model Checking Ansätze . . . 109
A.1.2. Type Checking Ansätze . . . 119
A.1.3. Static Analysis Ansätze . . . 127
A.1.4. Theorem Proving Ansätze . . . 131
A.2. Tabellen . . . 132
A.2.1. Tabellen zum Model Checking . . . 132

A.2.2. Tabellen zum Type Checking 140
A.2.3. Tabellen für die Untersuchung der Performanz bezüglich Static Analysis . . . 146
A.2.4. Tabellen für die Untersuchung der Performanz bezüglich Theorem Proving . 149
A.3. Quellcode des SPL für Mp3-Player 151

Abbildungsverzeichnis

2.1. Feature-Diagramm einer SPL für Mp3-Player 4
2.2. Vereinfachte Darstellung von Domain-Engineering und Application-Engineering als Teilprozesse des Software-Produktlinienparadigmas (orientiert an [46]) 5
2.3. Model Checking - Vereinfachte Darstellung 15
2.4. Drei Phasen des Model Checkings 16
2.5. Transitionssystem des MP3-Players inklusive des optionalen Features *RestartSong* des fortlaufenden Beispiels 20
2.6. Datenflussanalyse - Vereinfachte Darstellung einer möglichen Analyseprozedur (orientiert an [23]) 25

3.1. Beispiel der Produkt-basierten Strategie für das fortlaufende Mp3-Player-Beispiel . . 30
3.2. Beispiel der Familien-basierten Strategie für das fortlaufende Mp3-Player-Beispiel . 33
3.3. Beispiel der Produkt-basierten Strategie für das fortlaufende Mp3-Player-Beispiel . . 35
3.4. Darstellung der wissenschaftlichen Dokumente, die von Thüm et al. [79] berücksichtigt wurden und die Menge dieser Dokumente, die eine empirische Evaluation enthalten. 39
3.5. Anzahl evaluierter Software-Produktlinien der Model Checking Primärstudien . . . 40

4.1. Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit (a) der Anzahl der Features und (b) der Anzahl der Produkte 62
4.2. Effektgröße Cohen's d der einzelnen Primärstudien bezüglich der Verifikationszeit . 67
4.3. Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien mit (a) vielen Produkten und (b) wenigen Produkten 69
4.4. Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien mit (a) vielen Features und (b) wenigen Features 70
4.5. Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien in Abhängigkeit der Codegröße 70
4.6. Vergleich der explorierten Zustände beim (a) parallelen Produkt-basierten Model Checking und (b) sequenziellen Produkt-basierten Model Checking mit Familien-basierten Model Checking für die einzelnen Primärstudien 71
4.7. Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit (a) der Anzahl der Features und (b) der Anzahl der Produkte 76

4.8. Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien in Abhängigkeit der Codegröße 76
4.9. Effektgröße Cohen's d der einzelnen Type-Checking-Primärstudien bezüglich der Verifikationszeit 78
4.10. Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten für Software-Produktlinien mit (a) vielen Produkten und (b) wenigen Produkten 80
4.11. Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien mit (a) vielen Features und (b) wenigen Features 81
4.12. Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien mit mehr als 2000 Codezeilen 81
4.13. Darstellung der gesamten verfügbaren Daten zur Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit (a) der Anzahl der Features und (b) der Anzahl der Produkte 86
4.14. Darstellung der gesamten verfügbaren Daten zur Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit der Codegröße . 87
4.15. Darstellung der gesamten verfügbaren Daten zur Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit (a) der Anzahl der Features und (b) der Anzahl der Produkte 89
4.16. Darstellung der gesamten verfügbaren Daten zur Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit der Codegröße . 89

Tabellenverzeichnis

2.5. Vor- und Nachteile des Model Checkings 17
2.6. Tabelle der Eigenschaften, die mittels temporalen Logiken spezifizierbar sind 18

3.5. Eigenschaften und Vor- & Nachteile der drei primären Verifikationsstrategien 37
3.7. Verwendete wissenschaftliche Arbeiten zu Model Checking für die Evaluierung der Verifikationsstrategien 44
3.9. Verwendete wissenschaftliche Arbeiten zu Type Checking für die Evaluierung der Verifikationsstrategien 46
3.11. Verwendete wissenschaftliche Arbeiten zu Static Analysis für die Evaluierung der Verifikationsstrategien 47
3.13. Verwendete wissenschaftliche Arbeiten zu Theorem Proving für die Evaluierung der Verifikationsstrategien 48
3.15. Abhängige und unabhängige Variablen, die in dieser Arbeit betrachtet werden 49

4.2. Interpretation des d-Werts nach Cohen 58
4.4. Klassifizierung von Software-Produktlinien mit wenigen bzw. vielen Produkten, Features beziehungsweise Lines of Code 58
4.6. Abkürzungen für die Untersuchung der Hypothesen 60
4.8. Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die einzelnen Primärstudien in Bezug auf signifikanter Verbesserung der Verifikationszeit der Familien-basierten Strategie verglichen mit der Produkt-basierten Strategie 63
4.10. Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die miteinander verglichenen Verifikationsstrategien in der Primärstudie MC-A 10 bezüglich der Verifikationszeit 65
4.12. Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die weiteren verglichenen Verifikationsstrategien in den Primärstudien zum Model Checking bezüglich der Verifikationszeit 66
4.14. Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für den Speicherverbrauch der in den Primärstudien verglichenen Verifikationsstrategien zum Model Checking 66
4.17. Zusammenfassung der d_{Cohen}-Werte der einzelnen Primärstudien 68
4.19. Nullhypothesen, Alternativhypothesen und das Signifikanzniveau für die Hypothesen bezüglich Model Checking 68
4.21. Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die Hypothesen bezüglich Model Checking 69
4.22. Interpretation der Untersuchungsergebnisse bezüglich Model Checking 72
4.24. Gegenteilige Nullhypothesen und Alternativhypothesen sowie das Signifikanzniveau für die Hypothesen bezüglich Model Checking 73

4.26. Ergebnisse der Untersuchung der gegenteiligen Hypothesen mit dem Wilcoxon-Vorzeichen-Rang-Test . . . 73
4.28. Ergebnisse des Vote Countings der einzelnen Primärstudien in Bezug auf signifikanter Verbesserung der Verifikationszeit der Familien-basierten Strategie verglichen mit der Produkt-basierten Strategie . . . 77
4.30. Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die einzelnen Primärstudien bezüglich der weiteren miteinander verglichenen Strategien . . . 78
4.33. Zusammenfassung der d_{Cohen}-Werte der einzelnen Type-Checking-Primärstudien . . 79
4.35. Nullhypothesen, Alternativhypothesen und das Signifikanzniveau für die Hypothesen bezüglich Type Checking . . . 79
4.37. Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die Hypothesen bezüglich Type Checking . . . 80
4.38. Interpretation der Untersuchungsergebnisse bezüglich Type Checking . . . 82
4.40. Gegenteilige Nullhypothesen und Alternativhypothesen sowie das Signifikanzniveau für die Hypothesen bezüglich Model Checking . . . 83
4.42. Ergebnisse der Untersuchung der gegenteiligen Type-Checking-Hypothese mit dem Wilcoxon-Vorzeichen-Rang-Test . . . 84
4.44. Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests der Primärstudie SA-A 1 bezüglich einer signifikant geringeren Verifikationszeit der Familien-basierten Strategie verglichen mit der Produkt-basierten Strategie . . . 87
4.46. Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die einzelnen Theorem-Proving-Primärstudien . . . 90
4.48. Ergebnisse Hypothesenuntersuchungen der einzelnen Verifikationsmethoden . . . 96

A.1. Auflistung der Primärstudien zum Model Checking von Software-Produktlinien für den Vergleich der Verifikationsstrategien von Software-Produktlinien . . . 112
A.2. Auflistung der Primärstudien zum Type Checking von Software-Produktlinien für den Vergleich der Verifikationsstrategien von Software-Produktlinien . . . 123
A.3. Auflistung der Static-Analysis-Primärstudien von Software-Produktlinien für den Vergleich der Verifikationsstrategien von Software-Produktlinien . . . 129
A.4. Auflistung der Primärstudien zum Theorem Proving von Software-Produktlinien für den Vergleich der Verifikationsstrategien von Software-Produktlinien . . . 131
A.5. Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die Verifikationszeit der verglichenen Strategien der Primärstudie MC-A 10 . . . 134
A.7. PROD & FAM - Mittelwerte und Standardabweichungen der einzelnen Primärstudien, Effektstärke d_{Cohen} & das 95 % Konfidenzintervallen . . . 135
A.8. Tabelle für die Hypothese GH1 für die Verifikationszeit für viele Produkte . . . 136
A.9. Tabelle für Hypothese GH2 für die Verifikationszeit für wenige Produkte . . . 137
A.10. Tabelle für die Hypothese GH3 für die Verifikationszeit für viele Features . . . 138
A.11. Tabelle für die Hypothese GH4 für die Verifikationszeit für wenige Features . . . 138
A.12. Tabelle für die Hypothese GH5 für die Verifikationszeit in Abhängigkeit der Anzahl Codezeilen . . . 139

A.13. Tabelle für die Hypothese GH6 bezüglich Speicherverbrauch von parallelem Produkt-basierten Model Checking und Familien-basierten Model Checking 139
A.14. Tabelle für die Hypothese GH7 bezüglich Speicherverbrauch von sequenziellen Produkt-basierten Model Checking und Familien-basierten Model Checking 139
A.16. Auflistung der R-Befehle zur Ausführung der Wilcoxon-Vorzeichen-Rang-Tests für die jeweilige Hypothese bezüglich Model Checking. 140
A.18. Mittelwerte und Standardabweichungen der Produkt-basierten und Familien-basierten Strategie der einzelnen Type-Checking-Primärstudien , berechneter d_{Cohen}-Wert inklusive den 95 % Konfidenzintervallen 141
A.20. Mittelwerte und Standardabweichungen der weiteren miteinander verglichenen Strategien der einzelnen Type-Checking-Primärstudien , berechneter d_{Cohen}-Wert inklusive den 95 % Konfidenzintervallen 141
A.21. Tabelle für die Hypothese GH1 für die Verifikationszeit für viele Produkte 143
A.22. Tabelle für die Hypothese GH2 für die Verifikationszeit für wenige Produkte 143
A.23. Tabelle für die Hypothese GH3 für die Verifikationszeit für viele Features 144
A.24. Tabelle für die Hypothese GH4 für die Verifikationszeit für wenige Features 145
A.25. Tabelle für die Hypothese GH5 für die Verifikationszeit in Abhängigkeit der Anzahl Codezeilen 145
A.26. Tabelle der Verifikationszeit der Produkt-basierten Strategie für die Meta-Diagramme aus Kapitel 4.4 146
A.27. Tabelle der Verifikationszeit der Familien-basierten Strategie für die Meta-Diagramme aus Kapitel 4.4 149
A.28. Tabelle der Verifikationszeit der Produkt-basierten Strategie für die Meta-Diagramme aus Kapitel 4.5 150
A.29. Tabelle der Verifikationszeit der Familien-basierten Strategie für die Meta-Diagramme aus Kapitel 4.5 150

1 Einleitung

Software-Produktlinien haben sich in den letzten Jahren im Bereich der Software-Entwicklung sehr verbreitet. Eine Software-Produktlinie ermöglicht die Entwicklung ähnlicher Produkte, die sich in Teilen der bereitgestellten Funktionalitäten aufgrund von individuellen Kundenanforderungen unterscheiden können. Diese Unterschiede zwischen den einzelnen Produkten heißen Variabilität. Neben dieser Variabilität bieten Software-Produktlinien Vorteile wie geringere Entwicklungszeit (*Time-to-Market*) und geringere Entwicklungskosten [72].
Die Unterschiede und Gemeinsamkeiten der Produkte werden normalerweise mit Feature-Modellen repräsentiert. Ein Feature ist ein Merkmal oder eine Funktionalität eines Produkts, die für den Anwender sichtbar ist. Ein Feature-Diagramm ist eine graphische Repräsentation der Features einer Software-Produktlinie und deren Beziehungen zueinander [48].
Software-Produktlinien haben sich in einer Vielzahl an Domänen verbreitet wie beispielsweise im sicherheitskritischen Bereich, in eingebetteten Systemen und im Automotiv-Bereich [79]. Eine Domäne bezeichnet ein spezifisches Anwendungsgebiet. Beispiele für Software-Produktlinien in der Industrie sind z.B. die Drucker-Firmware von Hewlett-Packard (HP), Kontrollsysteme für Fahrstühle (*Elevetor Control System - ECS*) von LG oder Software für Fernsehgeräte und Audioanlagen von Philips [46]. Ein weiteres populäres Beispiel einer Software-Produktlinie ist das Betriebssystem *Linux*, das mehr als 10.000 Features zur Individualisierung besitzt [73].
Die verschiedenen Produkte einer Software-Produktlinie müssen bestimmte Korrektheitsanforderungen erfüllen. Dazu können formale Verifikationsmethoden verwendet werden. Formale Verifikation ermöglicht einen Beweis, dass ein System konform zu einer Spezifikation ist. Dazu werden mathematische Techniken und Methoden verwendet. Es existieren unterschiedliche statische Analysetechniken wie Model Checking [18], Type Checking [12], Static Analysis[14] und Theorem Proving[39]. Ursprünglich wurden diese Verifikationstechniken für einzelne Systeme entwickelt und sind nicht ohne Weiteres auf variable Entwicklungsparadigmen wie Software-Produktlinien anwendbar, da die Variabilität weitere Hindernisse wie z.B. höhere Verifikationszeit erzeugt. Darum wurden in den letzten Jahren viele Ansätze veröffentlicht, in denen statische Analysetechniken für Software-Produktlinien angepasst wurden (z.B [8, 65, 77]).
Software-Produktlinien bieten die Möglichkeit, statische Verifikationsmethoden auf unterschiedliche Art und Weise anzuwenden. Eine Möglichkeit ist eine Verifikationsmethode auf jedes Produkt einer Software-Produktlinie anzuwenden, was als *Produkt-basierte Strategie* bezeichnet wird. Außerdem kann jedes Feature isoliert verifiziert werden mit der *Feature-basierten Strategie*. Die *Familien-basierte Strategie* wird auf alle validen Kombinationen der Features einer Software-Produktlinie angewendet und wird dazu in ein Metaprodukt umgewandelt. Zusätzlich existieren Kombinationen dieser drei Strategien wie beispielsweise die *Feature-Produkt-basierte Strategie*. Thüm et al. haben diese Strategien klassifiziert [79] und eine Vielzahl von Analysemethoden für Software-Produktlinien in die Kategorien eingeordnet. Des Weiteren Thüm et al. haben diese Strategien bezüglich ihrer unterschiedlichen Vor- und Nachteile diskutiert [79].

Eine Software-Produktlinie besitzt bei n Features bis zu 2^n Produkte. Wenn bei einer Produkt-basierten Strategie die Verifikation auf jedes Produkt angewendet wird, werden dabei oftmals redundante Berechnungen ausgeführt, da sich die Produkte meistens nur in wenigen Features unterscheiden. Die Feature-basierte Strategie hingegen vermeidet redundante Berechnungen, kann aber nur lokale Eigenschaften beweisen und keine Eigenschaften, die sich auf mehrere Features beziehen. Viele der von Thüm et al. betrachteten Analysemethoden (z.B [7] oder [8]) wurden in Evaluierungen auf Eigenschaften wie Laufzeit und Speicherverbrauch untersucht. Die Evaluierungen enthielten unterschiedliche Software-Produktlinien mit unterschiedlicher Anzahl der Features und Anzahl der Produkte sowie divergierender Codegröße (LoC).
Idealerweise sollte für die Verifikation einer Software-Produktlinie die geeignetste Strategie verwendet werden. Bisher existieren wenige Arbeiten, die dieses Problem betrachten. Die Verifikationsmethoden sind schwierig miteinander vergleichbar aufgrund der in den Evaluierungen verwendeten unterschiedlichen Software-Produktlinien. Diese Arbeit vergleicht die Evaluierungen der recherchierten Analysemethoden und ermittelt sowohl abhängige als auch unabhängige Variablen einer Software-Produktlinie. Auf Basis dieser Vergleiche wird ermittelt, welche Analysestrategie für welche Kategorie von Software-Produktlinien signifikant besser oder schlechter ist. Diese Kategorie bezeichnet eine Menge von Software-Produktlinien mit bestimmter Codegröße, spezifischer Anzahl an Features oder Produkten. Um zukünftige Analysetechniken besser vergleichen zu können, werden außerdem Richtlinien für zukünftige Evaluierungen festgelegt. Zum einen soll die vorliegende Arbeit somit einen Vergleich der verschiedenen Verifikationsstrategien bieten, damit in der Praxis bei der Anwendung von Verifikationstechniken für Software-Produktlinien die beste Strategie für eine spezifische Software-Produktlinie ausgewählt werden kann. Zum anderen sollen die formulierten Richtlinien dieser Arbeit weitere zukünftige Vergleiche der Verifikationsstrategien angeregt und vereinfacht werden.

1.1. Gliederung

Die folgende Arbeit ist wie folgt aufgebaut. Das Kapitel 2 enthält eine Einführung in die notwendigen Grundlagen dieser Arbeit. Dazu zählen sowohl Software-Produktlinien als auch formale Verifikationsgrundlagen. Leser, die mit diesen Themen vertraut sind, können dieses Kapitel überspringen und bei Kapitel 3 fortfahren. In Kapitel 3 werden die verschiedenen Verifikationsstrategien vorgestellt und die Problemstellung erläutert. Des Weiteren enthält das Kapitel eine Darstellung der betrachteten Publikationen, die untersuchten Hypothesen bezüglich der Angemessenheit einer Strategie für bestimmte Software-Produktlinien sowie eine Erklärung, wie die Hypothesen analysiert wurden. In Kapitel 4 werden die Evaluierungen der Publikationen miteinander verglichen und die Ergebnisse werden mit Diagrammen und Tabellen dargestellt. In diesem Kapitel wird erläutert, welche der Hypothesen als wahr oder falsch erkannt wurden. Die Empfehlungen für zukünftige Evaluierungen werden in Kapitel 5 vorgestellt. Zu dieser Studie verwandte Arbeiten werden in Kapitel 6 präsentiert. In Kapitel 7 werden die errungenen Erkenntnisse rekapituliert und zukünftige Fragen und mögliche wissenschaftliche Arbeiten spezifiziert.

2 Grundlagen

In diesem Kapitel werden die benötigten Grundlagen für diese Arbeit vorgestellt. In Kapitel 2.1 werden Software-Produktlinien vorgestellt. Kapitel 2.2 bietet eine Einführung in die statischen Verifikationsmethoden Model Checking, Type Checking, Static Analysis und Theorem Proving.

2.1. Generierung von Software-Varianten

In den letzten Jahren ist die Nachfrage nach Software-Produkten stark gestiegen, die sich auf die individuellen Wünsche verschiedener Kunden anpassen lassen. Traditionell wurde für jeden Kunden eine neue Software-Variante von Grund auf neu entwickelt, wodurch hohe Kosten und lange Entwicklungszeiten entstanden [46]. Heutzutage können Software-Produkte in Standard-Software und individuelle Software unterteilt werden. Standard-Software sind Software-Produkte, die keine ausführliche Individualisierung unterstützen und für eine große Anzahl an Kunden entwickelt werden. Ein Beispiel für ein Standard-Softwareprodukt ist *Microsoft Word* [73]. Individuelle Software hingegen ist auf die Wünsche und Anforderungen eines einzelnen Kunden angepasst. Die Entwicklung eines individuellen Software-Produkts von Grund auf zeichnet sich durch hohe Kosten aus. Software-Produktlinien bieten die Möglichkeit eine Menge von individuellen Software-Produkten zu erstellen, die durch Wiederverwendung von gemeinsamen Programmartefakten die Entwicklungszeit und Kosten senken [46].

2.1.1. Software-Produktlinien

> "A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [1]

Software-Produktlinien bilden somit eine Menge von Software-Produkten, die eine Menge an Features teilen und für eine spezifische Domäne bestimmt sind. Diese Software-Produkte werden mittels einer vorher geplanten Vorgehensweise auf Basis dieser gemeinsamen wiederverwendbaren Teile gebildet [73]. Diese Produkte teilen einige Features und unterscheiden sich in anderen Features. Diese Unterschiede zwischen den einzelnen Produkten einer SPL werden als Variabilität bezeichnet [46].

> "A feature is an end-user visible characteristic of a system." [46]

Ein Feature ist eine für den Nutzer sichtbare Funktionalität des Systems. Ein Feature-Diagramm ist eine graphische Modellierungsmöglichkeit, um die Features einer SPL und deren Beziehungen darzustellen [48]. Dazu wird eine hierarchische Struktur verwendet, in der ein Feature-Diagramm immer genau einen Wurzelknoten und darunter weitere beliebige Ebenen von Features besitzt. Es wird zwischen abstrakten und konkreten Features unterschieden [73]. Ein abstraktes Feature dient

zur Gruppierung der Kind-Features und modelliert keine eigene Funktionalität, wohingegen konkrete Features eine tatsächliche Funktionalität repräsentieren. Features können optional oder obligatorisch (engl. mandatory) sein. Die gruppierten Features können unterschiedliche Beziehungen zu dem darüber liegenden abstrakten Feature besitzen. Sie können entweder eine Oder-Beziehung oder eine Alternative-Beziehung besitzen [73]. Eine solche Oder-Beziehung (*OR*) bezeichnet eine Gruppe von Features, bei der mindestens ein Feature der Gruppe ausgewählt werden muss. Eine Alternative-Beziehung legt fest, dass genau ein Feature der Gruppe ausgewählt werden muss. Des Weiteren besitzen Features eine Eltern-Kind-Relation (engl. *parent-child-relationship*), welche spezifiziert, dass bei Auswahl eines Kind-Features das darüberliegende Eltern-Feature obligatorisch ist. Zusätzlich sind noch weitere Abhängigkeiten zwischen nicht direkt verbundenen Features modellierbar, so genannte *Cross-Tree-Constraints* [73]. Ein Feature kann andere Features erzwingen oder ausschließen. Diese Relationen werden mittels aussagenlogischer Formeln spezifiziert.

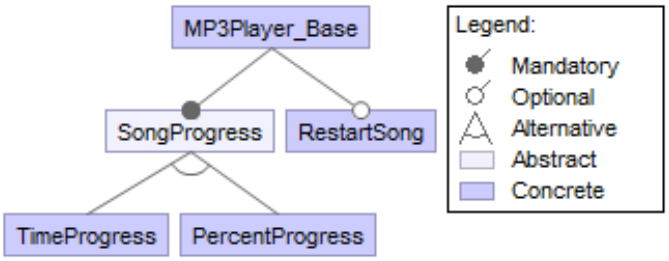


Abbildung 2.1.: Feature-Diagramm einer SPL für Mp3-Player

Die Abbildung 2.1 zeigt das Feature-Diagramm einer Software-Produktlinie für Mp3-Player. Dieses Feature-Diagramm stellt das fortlaufende Beispiel der vorliegenden Arbeit dar. Das Feature-Diagramm besitzt ein obligatorisches abstraktes Feature *SongProgress*, welches entweder durch das konkrete Feature *TimeProgress* oder das konkrete Feature *PercentProgress* realisiert werden kann. Ersteres stellt die Fortschrittsanzeige in Minuten und Sekunden dar, letzteres in Prozent. Des Weiteren besitzt das Feature-Diagramm das optionale Feature *RestartSong*, womit optional ein Musiktitel erneut abgespielt werden kann. Das Feature-Diagramm enthält keine Cross-Tree-Constraints.
Software-Produktlinien besitzen das Problem der Feature-Interaktionen. Eine Feature-Interaktion bedeutet, dass ein Feature isoliert fehlerfrei funktioniert, jedoch in Kombination mit einem oder mehreren anderen Features unerwartetes Verhalten erzeugt [79]. Der eigentliche Entwicklungsprozess des Software-Produktlinienparadigmas ist in zwei Teilprozesse unterteilt, dem *Domain-Engineering* und dem *Application-Engineering*.

> "Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realised." [46]

Im Domain-Engineering werden die Gemeinsamkeiten und Unterschiede der Produkte einer SPL analysiert, mit Hilfe von Feature-Diagrammen modelliert und durch eine Implementierungsmethode realisiert.

> "Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability." [46]

Im Application-Engineering werden die eigentlichen Produkte der SPL erzeugt. Dazu werden für ein spezifisches Produkt die gewünschten Features ausgewählt und unter Verwendung eines Generators das Produkt erzeugt. Die Abbildung 2.2 stellt diese beiden Teilprozesse nochmal graphisch dar.

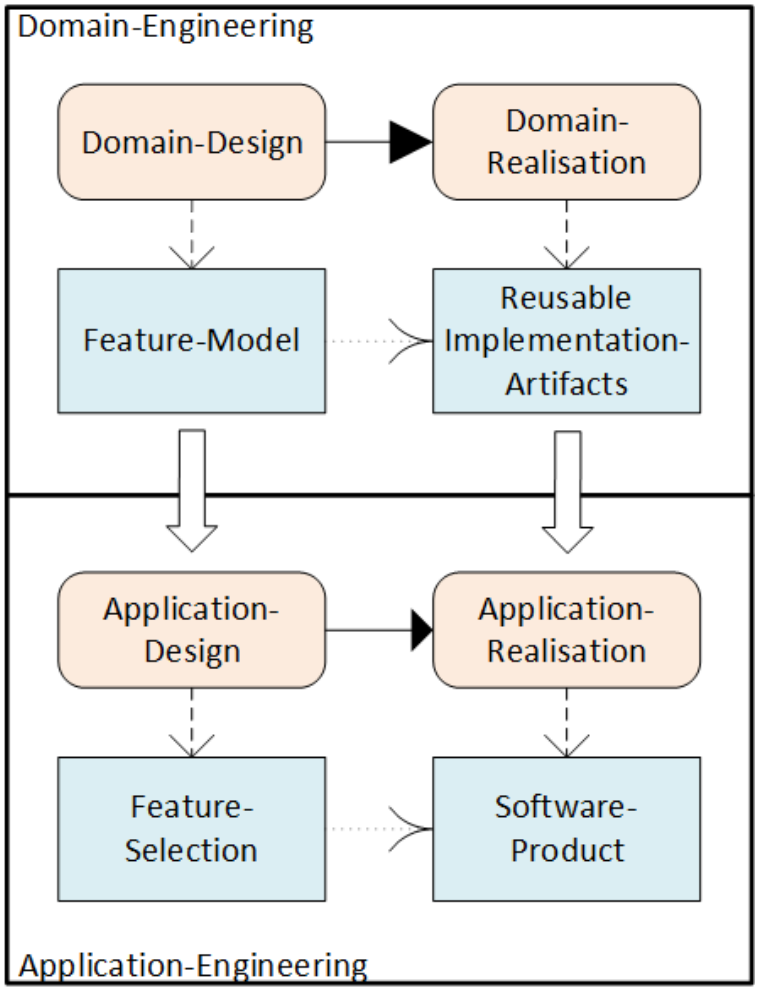


Abbildung 2.2.: Vereinfachte Darstellung von Domain-Engineering und Application-Engineering als Teilprozesse des Software-Produktlinienparadigmas (orientiert an [46])

Software-Produktlinien zeichnen sich durch einige Vorteile aus, welche zur Verbreitung des Software-Produktlinienansatzes beigetragen haben [72]. Zu diesen Vorteilen zählen:

- Verringerte Entwicklungskosten - Software-Produktlinien weisen anfängliche Mehrkosten für die Planung und Entwicklung der wiederverwendbaren Artefakte auf. Ab einer Menge

von 3-4 Produkten einer SPL amortisieren sich diese Kosten jedoch und SPL weisen deutliche Kosteneinsparungen auf [72].

- Kürzere Entwicklungszeiten (Time-to-Market) - Die Entwicklung der wiederverwendbaren Artefakte erzeugt eine zusätzliche Entwicklungszeit. Ist die Implementierung der Artefakte abgeschlossen, hat die Generierung der Produkte eine deutlich geringere Entwicklungszeit als die Entwicklung eines komplett neuen Produktes auf traditioneller Art und Weise [72].
- Höhere Qualität - Die Artefakte werden in vielen Produkten einer Software-Produktlinie verwendet. Das erhöht die Chance auf die Entdeckung von Fehlern, wodurch die Qualität erhöht wird [72].

2.1.2. Implementierung der Variabilität

Die Variabilität von Software-Produktlinien kann mit verschiedenen Techniken implementiert werden. Diese unterschiedlichen Ansätze können in Sprach-basierte Ansätze (engl. language-based approaches) und Werkzeug-basierte Ansätze (engl. tool-based approaches) unterteilt werden [73].

Definition 1. *"A language-based approach uses the mechanisms provided by a host programming language to implement features and to derive products. A tool-based approach uses one or more external tools to implement or represent features in code and to control the product-derivation process." [73]*

Sprach-basierte Ansätze verwenden Konzepte, die auf einer bestimmten Programmiersprache zur Implementierung von Features und zur Generierung von Produkten aus wiederverwendbaren Features basieren [73]. Werkzeug-basierte Implementierungstechniken verwenden hingegen ein Werkzeug oder mehrere, um Features als Codeartefakte zu erstellen und die Generierung der Produkte zu steuern [73]. Eine weitere Klassifizierungsmöglichkeit ist eine Unterteilung der Ansätze in Annotations-basierte (engl. annotation-based) und Kompositions-basierte (engl. composition-based) Techniken. In Annotations-basierten Ansätzen sind das Basisprogramm und alle Features in einer gemeinsamen Quellcode-Komponente aggregiert [73]. Annotationen innerhalb dieses Quellcodes spezifizieren, welche Codefragmente zu welchem Feature gehören.

Definition 2. *"Composition-based approaches implement features in the form of composable units, ideally one unit per feature. During product derivation, all units of all selected features and valid feature combinations are composed to form the final product." [73]*

Kompositions-basierte Techniken separieren den Quellcode eines Features oder einer Feature-Kombination innerhalb einer Datei oder eines Moduls [73]. Diese kompositionellen Einheiten werden bei der Generierung zu einem fertigen Produkt zusammengesetzt.
In dieser Arbeit werden nur solche Ansätze berücksichtigt, die eine automatische Generierung der Produkte erlauben. Dadurch werden beispielsweise Implementierungstechniken wie Versionsverwaltungssystem ausgeschlossen. Eine Einführung dieser ausgeschlossenen Implementierungstechniken ist in dem Buch von Apel et al. [73] zu finden. Die in dieser Arbeit berücksichtigten Implementierungsmethoden werden in den folgenden Abschnitten vorgestellt.

Frameworks

Frameworks bestehen aus einer Menge von Klassen, die eine Basis-Implementierung bilden und spezifische Erweiterungspunkte (*Hot Spots*), an denen die Basis-Implementierung um spezifische Funktionalität erweitert werden kann.

Definition 3. *"A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes. A framework is open for extension at explicit hot spots." [73]*

Ein Beispiel für ein Software-Framework ist die Entwicklungsumgebung Eclipse[1]. Eclipse ist eine ursprünglich für die Programmiersprache Java[2] vorgesehene Entwicklungsumgebung, die durch verschiedene, installierbare Plugins um weitere Features erweitert werden kann. Dadurch kann mit Eclipse beispielsweise zusätzlich die Erzeugung von UML[3] Diagrammen unterstützt werden.
Frameworks können noch in zwei Kategorien unterteilt werden, in *White-Box Frameworks* und *Black-Box Frameworks* [73]. White-Box Frameworks bestehen aus einer Menge von abstrakten und konkreten Klassen [73]. Die Änderung von Funktionalität wird durch Unterklassen realisiert. Diese Unterklassen können Methoden existierender Klassen ändern und überschreiben [73]. Der Name White-Box Framework resultiert daraus, dass der Entwickler das gesamte Programm verstehen muss, um eine Änderung durch eine Unterklasse zu realisieren. Die Kompilierung des eigentlichen Frameworks geschieht zusammen mit den Erweiterungen, wodurch ein Produkt zu generiert wird [73].
Black-Box Frameworks bestehen aus dem eigentlichen Framework (Basis-Programm) und Schnittstellen [73]. Eine Erweiterung wird als *Plug-In* bezeichnet [73]. Ein Plug-In muss die Schnittstelle implementieren, die den Hot Spot spezifiziert.

Definition 4. *"A plug-in extends hot spots of a black-box framework with custom behavior. A plug-in can be separately compiled and deployed." [73]*

Plug-Ins sind separat kompilierbar und können somit auch von Drittentwicklern implementiert werden. Normalerweise wird pro Feature ein Plug-In entwickelt. Bei den Black-Box Frameworks muss der Entwickler nur die Schnittstelle verstehen, anders als beim White-Box Framework.
Frameworks eignen sich gut zur Implementierung der Variabilität und bieten eine automatische Produktgenerierung durch Laden der spezifischen Plug-Ins [73]. Negative Eigenschaften des Framework-Ansatzes sind hoher Designaufwand vor der Implementierung und geringe Wiederverwendbarkeit außerhalb des Frameworks [73].

Die folgende Software-Produktlinie beschreibt beispielhaft die Implementierung von Variabilität durch den Ansatz eines Black-Box Frameworks. Hierbei handelt es sich um eine Software-Produktlinie für Mp3-Player. Jeder dieser möglichen Mp3-Player besitzt eine grundlegende Funktionalität, welche das Laden einer Mp3-Datei sowie Knöpfe zum Starten, Pausieren und Stoppen einer geladenen Mp3-Datei bietet. Die Variabilität dieser Software-Produktlinie besteht darin, dass die Fortschrittsanzeige der abgespielten Mp3-Datei entweder in realer Zeit oder in Prozent dargestellt werden kann. Außerdem kann dem Mp3-Player optional weitere Funktionalität in Form eines weiteren Knopfes hinzugefügt werden. In dem Beispiel wird als optionale zusätzliche Funktion der Neustart eines abgespielten Musiktitels implementiert. Das bereits vorgestellte Feature-Diagramm in Abbildung 2.1 stellt diese Software-Produktlinie dar, welche in dieser Arbeit als fortlaufendes Beispiel verwendet wird.
Listing 2.1 zeigt einen Teil des Black-Box Frameworks für die Mp3-Player Software-Produktlinie.

[1] https://eclipse.org/downloads/
[2] https://www.java.com/de/download/
[3] http://www.uml.org/

Die Basisfunktionalität, die dazugehörigen Objekte sowie grundlegende Layout-Methoden sind aus Platzgründen nicht dargestellt. Das Framework bietet unterschiedliche Konstruktoren, je nachdem welche Features bzw. Plug-ins in einem Produkt präsent sind. Der Konstruktor in Zeile 11 - 14 dient zur Initialisierung von Produkten, die eines der beiden alternativen Features *TimeProgress* und *PercentProgress* besitzen und in denen das optionale Feature *RestartSong* nicht verfügbar ist. Der Konstruktor in Zeile 16 - 22 erlaubt die Initialisierung eines Mp3-Players, der auch das optionale Feature *RestartSong* beinhaltet. In Zeile 31 wird die Methode *refreshStatus()* des Plug-ins für den Song-Fortschritt verwendet, um den Fortschritt der gewählten Mp3-Datei zu aktualisieren. In den Zeilen 37 - 42 wird unter Verwendung einer *if*-Bedingung geprüft, ob das Plug-in *RestartSong* präsent ist. In dem Fall wird der graphischen Oberfläche des Mp3-Players ein zusätzlicher Knopf hinzugefügt, der die Funktionalität des optionalen Plug-ins *RestartSong* durch einen Klick auslöst.

Listing 2.1: Blackbox Framework für die Software-Produktlinie eines MP3-Players

```
1  public class AMPGui {
2    private JFrame amp;
3    ...
4    JLabel progress;
5    ProgressIF progressPlugin = null;
6    AdditionalButtonIF additionalButtonPlugin = null;
7
8    public AMPGui() {
9    }
10
11   public AMPGui(ProgressIF progressPlugin) {
12     this.progressPlugin = progressPlugin;
13     progressPlugin.registerGui(this);
14   }
15
16   public AMPGui(ProgressIF progressPlugin,
17                 AdditionalButtonIF additionalButtonPlugin) {
18     this.progressPlugin = progressPlugin;
19     progressPlugin.registerGui(this);
20     this.additionalButtonPlugin = additionalButtonPlugin;
21     additionalButtonPlugin.registerGui(this);
22   }
23
24   public void createAndShowGui() {
25     ...
26     ActionListener playedTimeListener = new ActionListener() {
27       @Override
28       public void actionPerformed(ActionEvent argo) {
29           if (getMp3Player().getPlayerStatus() == 1) {
30           progress.setText(
31           progressPlugin.refreshStatus());
32           progress.revalidate();
```

```
33          }
34        }
35      };
36
37      if(additionalButtonPlugin != null) {
38        ...
39        JButton additionalButton = additionalButtonPlugin.getAdditionalButton();
40        additionalButtonPanel.add(additionalButton);
41        container.add(additionalButtonPanel);
42      }
43    }
44  }
```

Der Quellcode in Listing 2.2 dient als Schnittstelle für die Plug-in-Implementierungen der Features *TimeProgress* und *PercentProgress*. *TimeProgress* wird in Listing 2.3 dargestellt. Dieses Feature dient zur Anzeige des Abspiel-Fortschritts der geladenen Mp3-Datei als Zeit in Minuten und Sekunden. Feature *PercentProgress* stellt den Fortschritt in Prozent dar und die dazugehörige Implementierung als Plug-in ist in Listing 2.4 abgebildet. Beide Plug-ins implementieren die beiden Methoden *refreshStatus()* und *registerGui(AMPGui gui)* der Schnittstelle *ProgressIF*. Die Methode liefert den Fortschritt der abgespielten Mp3-Datei als Zeichenkette zurück. Die Methode *registerGui(AMPGui gui)* dient zur Initialisierung der graphischen Oberfläche, damit innerhalb der Plug-ins auf Objekte der graphischen Oberfläche zugegriffen werden kann.

Listing 2.2: Interface für das Plugin der Features TimeProgress und PercentProgress

```
1 public interface ProgressIF {
2   String refreshStatus();
3   void registerGui(AMPGui gui);
4 }
```

Listing 2.3: Plugin Implementierung des Features TimeProgress

```
1  public class TimeProgressPlugin implements ProgressIF{
2    AMPGui gui;
3
4    @Override
5    public String refreshStatus() {
6      long totalSongLength = gui.getMp3Player().getTotalSongTime();
7      long songPosition = gui.getMp3Player().getPlayedTime();
8      String songTime = convertTimeToString(songPosition);
9      String totalTime = convertTimeToString(totalSongLength);
10     return songTime + " / " + totalTime;
11   }
12
13   public String convertTimeToString(long microsecs) {
14     ...
15     String timeAsStr = ...
```

```
16      return timeAsStr;
17    }
18
19    @Override
20      public void registerGui(AMPGui gui) {
21        this.gui = gui;
22      }
23    }
```

Listing 2.4: Plugin Implementierung des Features PercentProgress

```
1  public class PercentProgressPlugin implements ProgressIF{
2    AMPGui gui;
3
4    public String refreshStatus() {
5      long totalSongLength = gui.getMp3Player().getTotalSongTime();
6      long songPosition = gui.getMp3Player().getPlayedTime();
7      double oneSecond = (double) (100.0 / totalSongLength);
8      double songPositionPercent = oneSecond * songPosition;
9      int songProgressPercent = (int) songPositionPercent;
10     return String.valueOf(songProgressPercent) + "%";
11   }
12
13   @Override
14   public void registerGui(AMPGui gui) {
15     this.gui = gui;
16   }
17 }
```

Listing A.1 zeigt die Schnittstelle *AddtionalButtonIF* für ein Feature, das der Benutzeroberfläche des Mp3-Players zusätzliche Funktionalität in Form eines Knopfes hinzufügen kann. Daher besitzt diese Schnittstelle eine Methodendeklaration *getAdditionalButton()*, die als Rückgabewert ein Objekt vom Typ *JButton* zurück liefert. Des Weiteren enthält die Schnittstelle die Methodendeklaration *registerGui(AMPGui gui)*, mit der das Objekt für die graphische Oberfläche des Mp3-Players im Plug-in gesetzt werden kann. Dem Button kann in der Plug-in-Implementierung eines Features ein *Action-Listener* zugeordnet werden, der die zusätzliche Funktionalität realisiert. Ein Beispiel für ein solches Plug-in ist die Implementierung des Features *RestartSong* in Listing 2.6. Das Feature ermöglicht das Neustarten einer Mp3-Datei, um sie erneut von Beginn an abzuspielen.

Listing 2.5: Interface für das Plugin des optionalen Features AdditionalButton

```
1 public interface AdditionalButtonIF {
2   JButton getAdditionalButton();
3   void registerGui(AMPGui gui);
4 }
```

Listing 2.6: Plugin Implementierung des Features RestartSong

```
1  public class RestartSong implements AdditionalButtonIF {
2    AMPGui gui;
3
4    @Override
5    public JButton getAdditionalButton() {
6      JButton restartTrack = new JButton("Restart Song");
7      restartTrack.addActionListener(new ActionListener() {
8        @Override
9        public void actionPerformed(ActionEvent argo) {
10         if (gui.getMp3Player() != null) {
11           if (gui.getPlayedTimer().isRunning()) {
12             gui.getMp3Player().restart();
13           }
14         }
15       }
16     });
17     return restartTrack;
18   }
19
20   @Override
21     public void registerGui(AMPGui gui) {
22     this.gui = gui;
23   }
24 }
```

In Listing 2.7 zeigt die Möglichkeit zur Ausführung des Frameworks unter Verwendung einer Hilfsklasse. Dabei werden in der *main*-Methode die Objekte *percentProgress* und *restartSong* deklariert, die vom Typ *ProgressIF* bzw. *AdditionalButtonIF* sind. Diese Objekte werden mit den Plug-ins *PercentProgressPlugin* und *RestartSong* initialisiert. Die Schnittstellen-Objekte werden dem Konstruktor für die graphische Oberfläche des Mp3-Players übergeben und damit die Benutzeroberfläche initialisiert. Anschließend wird im *try-catch*-Block die graphische Oberfläche gestartet und die Methode *createAndShowGui()* aufgerufen, die alle graphischen Elemente der Basisfunktionalität und der gewählten Features initialisiert . Üblicherweise wird ein Framework nicht auf diese Weise ausgeführt, indem die Plug-ins in der *main*-Methode initialisiert werden. Stattdessen werden alle zu einem Produkt gehörenden Plug-ins in einer *jar*-Datei archiviert, welche beim Starten des Frameworks geladen wird. Diese Methode wird in diesem Beispiel nicht verwendet, um das Beispiel einfach und intuitiv zu halten.

Listing 2.7: Hilfsklasse zum Ausführen des Mp3-Player Frameworks

```
1  public class FrameworkRunner {
2    public static void main(String[] args) {
3      ProgressIF percentProgress = new PercentProgressPlugin();
4      AdditionalButtonIF restartSong = new RestartSong();
5      final AMPGui gui = new AMPGui(percentProgress, restartSong);
6      try {
```

```
7         SwingUtilities.invokeAndWait(new Runnable() {
8           public void run() {
9             gui.createAndShowGui();
10          }
11        });
12      } catch ...
13    }
14  }
```

Der komplette Quellcode der Klassen, die zuvor nur teilweise vorgestellt wurden sowie die zusätzlichen Klassen der gesamten Software-Produktlinie, die zuvor nicht dargestellt wurden, sind im Anhang A in Abschnitt A.3 aufgeführt.

Weitere Implementierungstechniken

In den folgenden Abschnitten werden weitere mögliche Implementierungstechniken zusammenfassend erläutert.

Präprozessor-Annotationen Der Ansatz mittels Präprozessoren ist Werkzeug-basiert [73]. Präprozessoren bearbeiten den Quellcode vor seiner Kompilierung [73]. Dazu wird eine konditionelle Kompilierung genutzt, bei der mittels Präprozessor-Direktiven wie z.B. *#IFDEF* und *#ENDIF* Quellcodefragmente spezifiziert werden, die vor der Kompilierung entfernt werden sollen. Diese Entfernung stellt oftmals ein Auskommentieren der Codefragmente dar. Normalerweise wird pro Feature ein Name verwendet und mittels Makro-Definition dieser Name nur definiert, wenn das korrespondierende Feature im Produkt präsent ist [73]. Präprozessoren bieten oftmals auch negative Annotationen wie *#IFNDEF*, konditionelle Bedingungen wie *#ELSE* und Verschachtelung dieser Sprachkonstrukte [73]. Vorteil des Präprozessor-Ansatzes ist die Verfügbarkeit von Präprozessoren wie *CPP*[4] für die Programmiersprache C oder *ANTENNA*[5] für Java [73]. Außerdem ist das konditionelle Entfernen von Codefragmenten intuitiv und einfach anwendbar. Der Präprozessor-Ansatz erzeugt allerdings schwer verständlichen und schwer wartbaren Programmcode [73]. Des Weiteren unterstützt dieser Ansatz keine Trennung der einzelnen Funktionalitäten in separate Implementierungsartefakte [73].

Feature-Orientierte Programmierung (FOP) Feature-orientierte Programmierung ist ein Kompositions-basierter und Sprach-basierter Ansatz zur Implementierung der Variabilität [73]. Er verwendet direkt das Konzept der Features, wobei ein System in einzelne separate Features zerlegt wird [73]. Bestenfalls wird eine Systemkomponente pro Feature implementiert. Voraussetzung für die Feature-orientierte Programmierung ist die Abbildung von Features zu Implementierungsartefakten [73]. Bei der Feature-orientierten Programmierung wird ein Produkt nach der Feature-Auswahl automatisch zusammengesetzt.
Ein grundlegendes Konzept für die Feature-orientierte Programmierung sind Kollaborationen. Eine Kollaboration bezeichnet eine Menge von interagierenden Klassen, welche die Funktionalität eines Features implementieren [73]. Dabei spielt jede Klasse eine spezifische Rolle. Üblicherweise werden mehrere Features durch mehrere Kollaborationen implementiert [73]. Daher sind einzelne Klassen in mehreren Kollaborationen enthalten. Eine Rolle spezifiziert die Zuständigkeit einer

[4] https://gcc.gnu.org/onlinedocs/cpp/
[5] http://antenna.sourceforge.net/wtkpreprocess.php

Klasse in einer Kollaboration [73]. Die Abbildung von Kollaborationen zu Features wird als Feature-Modul bezeichnet [73]. Ein Produkt einer Software-Produktlinie wird durch Kombination verschiedener Feature-Module erzeugt. Ein Feature-Modul kann neue Elemente zu einem Programm hinzufügen oder existierende Elemente überschreiben und erweitern [73]. Dadurch wird das Verhalten des Basisprogramms verändert. Aufgrund dieser Schritt für Schritt Änderung ist die Reihenfolge der Feature-Module wichtig, damit z.B. existierende Elemente wie gewünscht verändert werden können [73]. Da Feature-orientierte Programmierung ein Sprach-basierter Ansatz ist, müssen existierende Sprachen zur Unterstützung dieses Ansatzes erweitert werden.

Aspekt-orientierte Programmierung(AOP) Aspekt-orientierte Programmierung ist, wie die Feature-orientierte Programmierung, ein Sprach- und Kompositions-basierter Implementierungsansatz [73]. Das Ziel von Aspekt-orientierter Programmierung ist die Modularisierung von Belangen, welche sich über einzelne Modulgrenzen erstrecken [73]. Diese Modulgrenzen-überschreitenden Belange werden *querschneidende Belange* (engl. *crosscutting concerns*) genann [73]t. Querschneidende Belange führen zu Problemen wie *verstreutem Code* (engl. *code scattering*) und *vermischtem Code* (engl. *code tangling*) [73]. Beim Code Scattering ist der Quellcode eines Belanges auf mehrere Implementierungsartefakte wie z.B. Module oder Plug-ins verteilt [73]. Code Tangling bezeichnet Implementierungsartefakte, die den Quellcode mehrerer Belange in einer einzelnen Komponente enthalten [73].
Aspekt-orientierte Programmierung zielt auf Verminderung von verstreutem und vermischtem Code. Dazu werden in der Aspekt-orientierten Programmierung *Aspekte* verwendet. Features, die mittels traditionellen Modularisierungstechniken wie Klassen ohne Code Scattering und Code Tangling implementiert werden können, werden auf die Weise implementiert. Querschneidende Belange hingegen werden mittels Aspekten realisiert [73]. Produkte werden im Aspekt-orientierten Ansatz mit einem *Aspekt-Weber* (engl. *aspect weaver*) zusammengefügt [73]. Dieses Werkzeug webt die Aspekte an dafür vorgesehenen Positionen ins Basisprogramm ein und erzeugt dadurch ein fertiges Produkt [73]. Diese vom Entwickler vorgesehenen Erweiterungspositionen werden *Join Points* genannt [73]. Diese Joint Points werden durch Ereignisse repräsentiert. Ein Ereignis ist die Ausführung einer Methode oder der Zugriff auf ein Feld. Weiterhin beschreibt ein Pointcut in einem Aspekt, welche Join Points von einem Aspekt genutzt werden [73]. Außerdem existiert ein Konstrukt namens *Advice*. Ein Advice ist ein Konstrukt zur Kapselung der Anweisungen, die durch eine Menge von Join Points ausgeführt werden und ähnelt somit einer Methode [73]. Durch Verwendung von Intertyp-Deklarationen können Methoden, Felder oder Schnittstellen eines Aspekts in eine existierende Klasse eingefügt werden.

Delta-orientierte Programmierung (DOP) Delta-orientierte Programmierung ist ähnlich zu der Aspekt-orientierten Programmierung. Dieses Paradigma ist ebenfalls Sprach- und Kompositions-basiert [73]. Es besteht aus einem Basisprogramm und beliebig vielen Delta-Modulen [73]. Die Delta-orientierte Programmierung kann wie die Aspekt-orientierte Programmierung neue Klassen, Felder und Methoden hinzufügen sowie Methoden überschreiben [73]. Zusätzlich können Delta-Module noch existierende Klassen, Methoden und Felder aus dem Programm entfernen [73]. Delta-orientierte Programmierung ermöglicht somit als Basisprogramm ein Programm, dass die gesamte Funktionalität enthält, welche zur Erzeugung verschiedener Produkte schrittweise entfernt werden kann. Dadurch wird bei der Auslieferung des Produkts nicht der gesamte Programmcode ausgeliefert, wie z.B. beim Präprozessor-Ansatz. Nachteilig ist bei der Delta-Orientierung die Verfolgung,

ob ein oder mehrere Delta-Module benötigte Programmelemente entfernen. Hierfür bietet Type Checking eine gute Möglichkeit dieses Problem zu beheben [73].

2.2. Statische Verifikation

Statische Verifikation (oder formale Verifikation) verwendet mathematische Argumente, um den Beweis der Einhaltung einer Spezifikation oder einer Teilmenge davon durch ein System herzuleiten [21]. Oftmals gibt es Unklarheiten mit den Begriffen Verifikation und Validierung. Diese beiden Begriffe bezeichnen unterschiedliche Dinge und sollten daher nicht vertauscht oder verwechselt werden. Validierung bezieht sich auf die Überprüfung, ob ein formales Modell der Design-Spezifikation entspricht ("check that we are verifying the right thing" [18]), das heißt ob das richtige System spezifiziert wurde. Verifikation hingegen behandelt die Prüfung, ob das implementierte System die spezifizierten Anforderungen erfüllt ("check that we are building the thing right" [18]). Verifikation bedeutet die Überprüfung, ob ein betrachtetes Programm bestimmte spezifizierte Eigenschaften erfüllt. Grundlage für die Verifikation ist die Spezifikation von Eigenschaften, die das System erfüllen soll. Ein Programm wird als korrekt betrachtet, wenn alle Spezifikationen erfüllt werden. Dabei wird deutlich, dass die Korrektheit eines Programms immer relativ zu den spezifizierten Eigenschaften ist. Ist die Spezifikation fehlerhaft und wird trotzdem von dem Programm eingehalten, bedeutet das nicht, dass sich das System fehlerfrei verhält [18]. Formale Verifikation dient zur Modellierung und Analyse von Programmen auf Basis von mathematischen Methoden. Verifikation ist sehr wichtig, um fehlerfreie Software-Produkte zu erzeugen und folgenreiche Fehler zu vermeiden. Ein populäres Beispiel für einen Software-Fehler ist der Absturz der Trägerrakete Ariane 5 im Jahr 1996, die sich aufgrund eines Speicherüberlaufs nach 37 Sekunden quer ausrichtete und sich selbst sprengte [60]. Dieser Fehler hätte durch formale Verifikationstechniken verhindert werden können [35]. In den nachfolgenden Abschnitten werden die formalen Verifikationstechniken Model Checking, Type Checking, Theorem Proving und Static Analysis vorgestellt. Dabei wird das Model Checking ausführlich erklärt, da zum Model Checking die meisten Ansätze verfügbar sind. Die anderen drei formalen Verifikationstechniken werden hingegen in kürzerer Form vorgestellt.

2.2.1. Model Checking

Model Checking ist eine Verifikationstechnik, bei der ein System bzw. das Verhalten eines Systems mittels Modellen dargestellt werden. Dabei sind diese Modelle mathematisch korrekt und eindeutig. Die Modell-basierte Verifikation ist abhängig von der Güte des Modells [18].

> "Any verification using model-based techniques is only as good as the model of the system." [18]

Beim Model Checking werden alle Zustände des modellierten Systems exploriert [18]. Diese Exploration aller Systemzustände erfolgt typischerweise auf einer Brute-Force-Methode. Der Model Checking Algorithmus wird von einem Werkzeug, dem so genannten Model Checker, ausgeführt. Dabei wird für alle Systemzustände überprüft, ob in diesen Zuständen die spezifizierten Eigenschaften eingehalten werden [18]. Somit wird das komplette Systemverhalten in Bezug zur Spezifikation verifiziert. Aufgrund der möglichen Unvollständigkeit der Spezifikation wird nie das vollständige, reale System verifiziert [18]. Zusätzlich existieren Methoden wie Wahrscheinlichkeits-basiertes Model Checking, welche anstatt des gesamten Zustandsraums nur eine Teilmenge davon explorieren

und damit die Genauigkeit des Verifikationsergebnisses verringern [18].
Der Prozess des Model Checkings wird in Abbildung 2.3 vereinfacht dargestellt. Zum einen werden anhand der Kundenanforderungen eine Anforderungsspezifikation erstellt. Daraus werden die gewünschten Eigenschaften extrahiert und spezifiziert. Zum anderen wird aus dem zu prüfenden Softwaresystem ein Systemmodell erzeugt. Die spezifizierten Eigenschaften sowie das Systemmodell bilden die Eingabe des Model Checkers, welcher die Gültigkeit der spezifizierten Eigenschaften verifiziert.

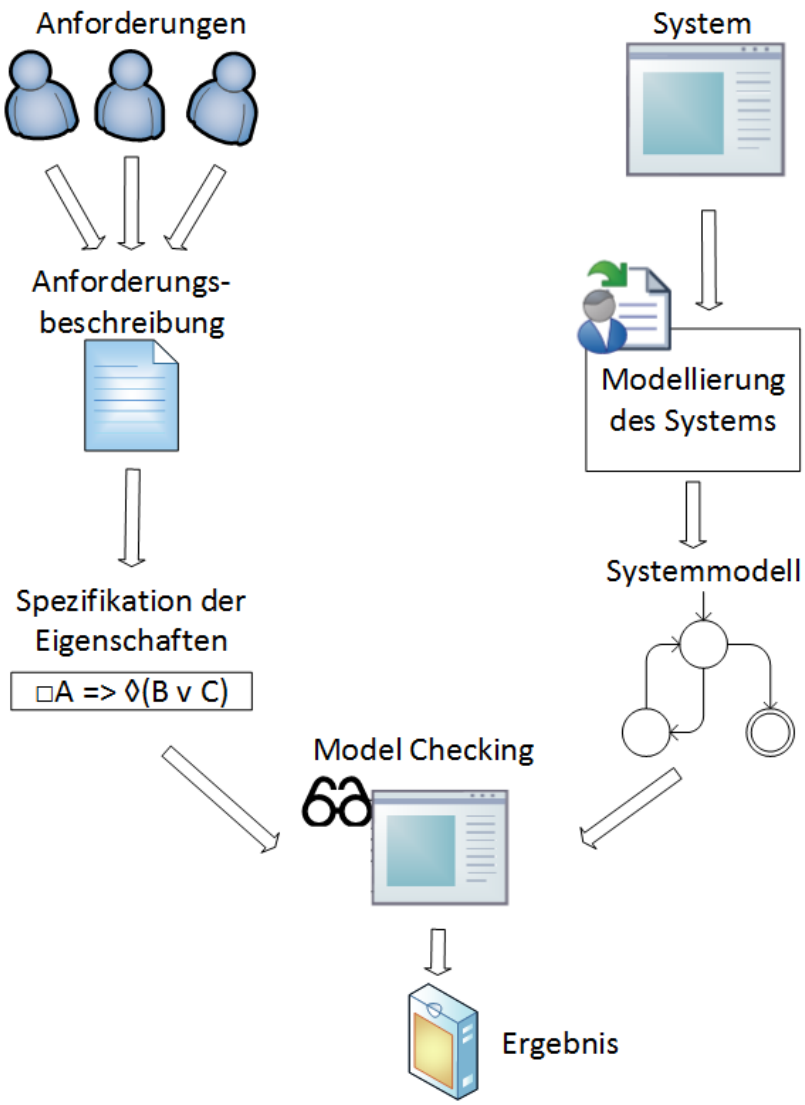


Abbildung 2.3.: Model Checking - Vereinfachte Darstellung

Eigenschaften, die mittels Model Checking verifiziert werden können, sind beispielsweise *Safety*- oder *Liveness*-Eigenschaften [79]. Dazu zählen unter anderem die Abwesenheit von Deadlocks oder Race Conditions. Eine *Safety-Eigenschaft* besagt, dass etwas Schlechtes niemals passieren darf. Was etwas Schlechtes bedeutet, ist von dem jeweiligen Programm abhängig. Beispielsweise darf ein Feature, dass ein weiteres Feature benötigt, niemals in einem Programm verfügbar sein, welches das

benötigte Feature nicht auch enthält. Eine *Liveness-Eigenschaft* besagt hingegen, dass etwas Gutes eventuell zukünftig geschieht. Beispielsweise könnte eine Liveness-Eigenschaft beschreiben, dass ein Programm zukünftig irgendwann eine Funktion ausführen kann und somit nicht in einem Deadlock hängen bleibt. Eine präzise und eindeutige Formulierung dieser Eigenschaften ist ebenfalls Voraussetzung für das Model Checking.
Häufig kann das Modell des Systems automatisch aus dem eigentlichen Programmcode generiert werden [18]. Dadurch wird die Verifikation vereinfacht, da der Software-Verifizierer nicht zusätzlich auch das Systemmodell erstellen muss. Model Checking benötigt sowohl ein Modell des Systems als auch eine Spezifikation der Eigenschaften [18]. Das Systemmodell beschreibt, wie sich das Programm verhält [18]. Dahingegen legt die Spezifikation der Eigenschaften fest, wie sich das Programm verhalten soll [18]. Wie schon erläutert, prüft der Model Checker für alle Systemzustände, ob die spezifizierten Eigenschaften eingehalten werden. Wird dabei ein Zustand entdeckt, der eine Eigenschaft verletzt, wird als Ergebnis der Verifikation ein Gegenbeispiel zurückgeliefert [18]. Dieses Gegenbeispiel ist ein Ausführungspfad, der die Reihenfolge der Zustände vom Anfangszustand bis zum dem Zustand beschreibt, in dem die Eigenschaft verletzt wird [18]. Dadurch ist es für den Software-Verifizierer möglich, die Verletzung der Eigenschaft zu untersuchen und zu beheben, indem die Fehlerquelle behoben wird. Die Fehlerquelle kann entweder im Programm liegen, wobei das Programm überarbeitet werden muss, um den Fehler zu beheben [18]. Oder eine andere Möglichkeit für die Verletzung der Eigenschaft kann durch eine fehlerhafte Spezifikation einer Eigenschaft entstehen [18]. Hierbei muss die Spezifikation verbessert werden. Oder der Fehler kann ein fehlerhaftes Systemmodell sein, welches das System nicht korrekt repräsentiert [18]. Dieser Fehler kann vor allem bei manuell extrahierten Systemmodellen entstehen. Wenn der Fehler im Systemmodell liegt, muss dieses überarbeitet werden.

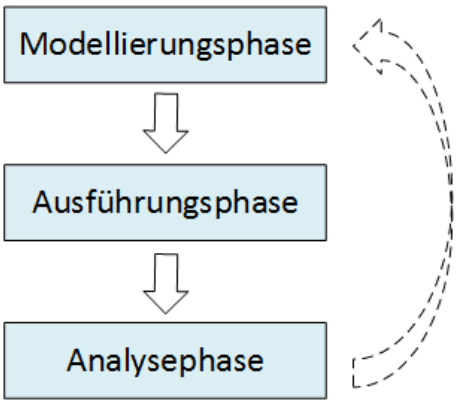


Abbildung 2.4.: Drei Phasen des Model Checkings

Model Checking besteht üblicherweise aus drei Phasen. Diese Phasen werden in Abbildung 2.4 dargestellt. In der Modellierungsphase (engl. modeling phase) wird zum einen das System modelliert und zum anderen die Spezifikation der Eigenschaft erstellt [18]. Zusätzlich ist in dieser Phase die Ausführung von Simulationen möglich, sofern ein Simulator vorhanden ist [18]. In der Ausführungsphase (engl. running phase) wird der Model Checker ausgeführt, um die Validität der spe-

zifizierten Eigenschaft zu prüfen. Dabei wird algorithmisch in jedem Zustand des Systemmodells verifiziert, ob die betrachtete Eigenschaft eingehalten oder verletzt wird [18]. Während der Analysephase (engl. analysis phase) wird das Ergebnis des Model Checkers analysiert [18]. Die folgenden vier Resultate sind möglich.

- Eigenschaft eingehalten
- Eigenschaft verletzt
- Speicher reicht nicht aus
- Timeout tritt ein

In dem Fall, dass die Eigenschaft gültig ist, kann eine weitere Eigenschaft geprüft werden [18]. Wenn alle Eigenschaften erfolgreich verifiziert wurden, wird die Verifikation beendet. Wenn eine Eigenschaft verletzt wird, muss das Gegenbeispiel analysiert werden und entweder das Programm, das Systemmodell oder die Spezifikation korrigiert werden [18]. Danach muss der gesamte Verifikationsprozess wiederholt werden. Sollte der Speicher nicht ausreichen, gibt es Möglichkeiten wie Abstraktion oder symbolisches Model Checking [9], um das Systemmodell bzw. den benötigten Arbeitsspeicher zu reduzieren [18]. Dazu werden z.B. Gleichmäßigkeiten in der Struktur des Systemmodells genutzt. Eine weitere Möglichkeit zur Reduzierung des Zustandsraums (engl. state space) ist die Verwendung von Wahrscheinlichkeits-basiertem Model Checking, wobei nur ein bestimmter Teil des Zustandsraums verifiziert wird. In diesem Fall wird die Genauigkeit des Verifikationsergebnisses reduziert, da nicht der gesamte Zustandsraum verifiziert wird [18]. Außerdem kann während des Model Checkings ein Timeout auftreten. Gründe dafür können nicht ausreichender Arbeitsspeicher oder eine Überschreitung der maximalen Verifikationszeit sein.
Model Checking zeichnet sich durch eine Vielzahl von Vor- und Nachteilen aus. Einige davon werden in der folgenden Tabelle 2.5 vorgestellt.

Vorteile	**Nachteile**
Für eine Vielzahl verschiedener Anwendungen einsetzbar [18]	Abhängig von der Qualität des Systemmodells [18]
Liefert ein Gegenbeispiel für die Fehlerdiagnose [18]	Manuelle Spezifikation notwendig, fehlerhaft oder unvollständig sein kann [18]
Automatisierbar [18]	*State-Space-Explosion*, wobei der benötigte Speicher den verfügbaren Arbeitsspeicher übersteigt [18]
Basiert auf formalen Definition und besitzt dadurch präzise Notationen [18]	Ungeeignet für datenintensive Systeme [18]

Tabelle 2.5.: Vor- und Nachteile des Model Checkings

Eine ausführlichere Auflistung der Vor- und Nachteile des Model Checkings ist in dem Buch von Baier und Katoen [18] zu finden.
Die Spezifikation der Eigenschaften eines Systems geschieht mittels einer präzisen und eindeutigen Spezifikationssprache. Beispiele für eine solche Spezifikationssprache sind temporale Logiken wie *Linear Temporal Logic (LTL)* oder *Computation Tree Logic (CTL)*. Temporale Logiken sind eine Erweiterung der Aussagenlogik um temporale Operatoren, mit denen das unendliche Verhalten eines Sys-

tems spezifiziert werden kann. Diese temporalen Operatoren ermöglichen Aussagen zum Verhalten eines Systems zu einem bestimmten Zeitpunkt. Ein solcher Zeitpunkt stellt keine reale Zeit dar, sondern spezifiziert die Reihenfolge von Zuständen, womit z.B. Eigenschaften modelliert werden können, die nach einem bestimmten Vorgängerzustand gelten sollen. Temporale Logiken besitzen eine präzise mathematische Notation und sind intuitiv anwendbar. Temporale Logiken erlauben die Spezifikation der folgenden Eigenschaftsklassen [18], die in Tabelle 2.6 dargestellt werden.

Eigenschaftsklasse	**Beispiel**
Funktionale Korrektheit	Entspricht das Systemverhalten dem erwarteten Verhalten? [18]
Safety	Es geschieht niemals etwas Schlechtes [18]
Liveness	Es geschieht eventuell etwas Gutes [18]
Erreichbarkeit (engl. Reachability)	Enthält das System einen Deadlock? [18]
Gerechtigkeit (engl. Fairness)	Ist ein wiederholtes Auftreten eines Ereignisses unter bestimmten Umständen möglich? [18]
Echtzeit-Eigenschaften	Werden Echtzeit-Anforderungen eingehalten?

Tabelle 2.6.: Tabelle der Eigenschaften, die mittels temporalen Logiken spezifizierbar sind

Temporale Logiken können auf zwei unterschiedlichen Zeitbetrachtungen basieren, der linearen Zeit (engl. *linear time*) und der verzweigenden Zeit (engl. *branching time*) [18]. Bei linearer Zeit existiert genau ein nachfolgender Zustand zu jedem Zeitpunkt im System. Mit Zeitpunkt ist nicht die Zeit im allgemeinen Sinne zu verstehen, sondern ein Zustand des Systems. Dies erlaubt eine Spezifikation einer Zustandsreihenfolge [18]. Verzweigende Zeit hingegen kann alternative Ausführungspfade enthalten, sodass ein Zeitpunkt bzw. Zustand mehr als einen nachfolgenden Zustand besitzt, wodurch eine baumartige Struktur entsteht.
Temporale Logiken erweitern die normale Aussagenlogik normalerweise um die folgenden temporalen Operatoren.

- $\Diamond$ *eventually* - Eine Eigenschaft gilt eventuell in einem zukünftigen Zustand
- $\Box$ *always* - Eine Eigenschaft gilt ab dem momentanen Zustand zukünftig immer

LTL basiert auf linearer temporaler Logik, d.h. in jedem Zeitpunkt existiert ein Folgezustand für den aktuellen Zustand.

Definition 5. *Syntax von LTL [18]*
Eine LTL Formel über einer Menge AP von atomaren Aussagen wird anhand folgender Grammatik gebildet:

$$\varphi := true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \bigcup \varphi_2$$

wobei $a \in AP$

$\varphi_1 \wedge \varphi_2$ ist die Konjunktion zweier LTL Formeln und $\neg\varphi$ die Negation einer LTL Formel. $\bigcirc\varphi$ ist der *Next-Operator*, wobei $\bigcirc\varphi$ im aktuellen Zeitpunkt gilt, falls φ im nächsten Zustand gilt. Analog ist die Semantik des *Until-Operator* $\varphi_1 \bigcup \varphi_2$. Dabei gilt $\varphi_1 \bigcup \varphi_2$ im aktuellen Zustand, wenn ein zukünftiger Zustand existiert, in dem φ_2 gilt und bis zu diesem Zustand in jedem vorherigen Zustand φ_1 gilt. LTL erlaubt die Spezifikation von *Fairness*-Annahmen ohne weitere Techniken oder Erweiterungen. Fairness wird unterschieden zwischen starker Fairness (engl. *strong fairness*) und schwacher

Fairness (engl. *weak fairness*). Starke Fairness besagt, dass jeder mögliche Prozess unendlich oft bearbeitet wird. Dahingegen bedeutet schwache Fairness, dass jeder Prozess, der ab einem bestimmten Zeitpunkt verfügbar ist, unendlich oft bearbeitet wird [18].
Computation Tree Logik (CTL) hingegen basiert auf verzweigender Zeitbetrachtung. LTL ermöglicht Aussagen über Eigenschaften, die für alle Berechnungen gelten sollen, die bei einem bestimmten Startzustand beginnen. Dabei sind jedoch keine Aussagen über einige dieser Berechnungen möglich. CTL hingegen ermöglicht diese Betrachtung von einigen dieser Berechnungen. Eine Beispiel-Eigenschaft dafür sieht wie folgt aus.

> "For every computation it is always possible to return to the initial state." [18]

Definition 6. *Syntax von CTL [18]*
Eine CTL Formel über einer Menge AP von atomaren Aussagen wird anhand folgender Grammatik gebildet:

$$\Phi ::= true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

wobei $a \in AP$ und φ eine Pfadformel ist.
Eine CTL Pfadformel basiert auf folgender Grammatik:

$$\varphi ::= \ \mid \bigcirc \Phi \mid \Phi_1 \bigcup \Phi_2$$

wobei Φ, Φ_1 und Φ_2 Zustandsformeln sind.

Die aus LTL bekannten Quantoren und Grammatikbestandteile haben dieselbe Bedeutung wie bei LTL. Der *Exist-Quantor* ($\exists$) erlaubt eine Aussage über die Existenz eines Pfad, für den eine spezifizierte Eigenschaft gilt. Der *All-Quantor* ($\forall$) hingegen spezifiziert eine Eigenschaft, die in allen Ausführungspfaden gilt.

Modellierung von Systemen

Der Model Checker erwartet als Eingabe unter anderem ein Modell, welches das Verhalten des Systems in einer präzisen und eindeutigen Form beschreibt. Dazu existieren mehrere Modell-Klassen wie z.B. Transitionssysteme (engl. *transition systems*) [18]. Ein Transitionssystem (*TS*) ist ein gerichteter Graph bestehend aus Knoten und Kanten. Die Knoten modellieren die möglichen Zustände des Systems und die Kanten die Übergänge zwischen Zuständen [18]. Die Kanten werden mit Namen für Aktionen (engl. *actions*) und die Zustände mit atomaren Aussagen annotiert. Ein Aktionsname beschreibt das Ereignis, welches den Übergang von einem Zustand zu einem anderen Zustand auslöst. Eine atomare Aussage ist eine aussagenlogische Formel, die sich entweder zu *wahr* oder *falsch* auswerten lässt und nicht in kleinere Aussagen zerlegt werden kann.

Definition 7. *Transitionssystem (TS)*
Ein Transitionssystem TS ist ein Tupel $(S, Act, \longrightarrow, I, AP, L)$ mit

S	*Menge von Zuständen*
Act	*Menge von Aktionen*
$\longrightarrow \subseteq S \times Act \times S$	*als Transitionsrelation*
$I \subseteq S$	*Menge von Startzuständen*
Ap	*Menge von atomaren Aussagen*
$L : S \rightarrow 2^{AP}$	*Beschriftungsfunktion*

Die Abbildung 2.5 zeigt das Beispiel eines Transitionssystems. Dieses Transitionssystem modelliert das Produkt inklusive dem optionalen Feature *RestartSong* des fortlaufenden Beispiels der Software-Produktlinie für MP3-Player aus Kapitel 2.1.2.

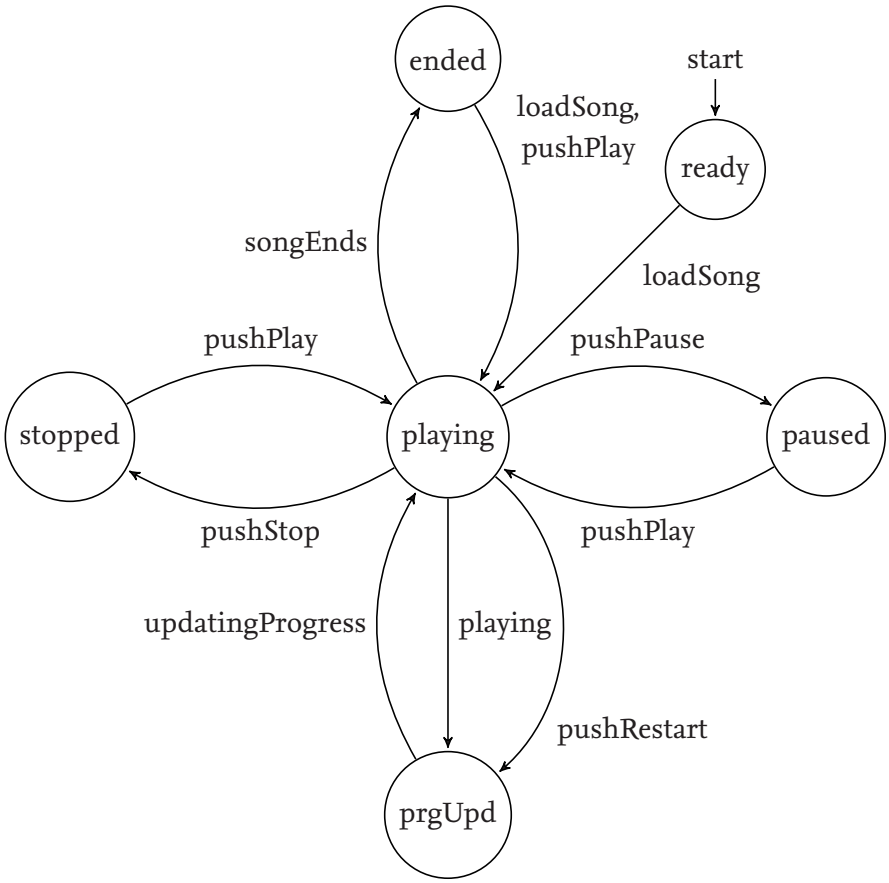


Abbildung 2.5.: Transitionssystem des MP3-Players inklusive des optionalen Features *RestartSong* des fortlaufenden Beispiels

Transitionssysteme und andere Systemmodell-Arten wie der *Calculus of Communicating Systems* (*CCS*) [29] sind zur Modellierung von einzelnen Systemen geeignet, jedoch nicht zur Modellierung der Variabilität von Software-Produktlinien. Für Software-Produktlinien wurden viele dieser Modellierungstechniken um die Repräsentation von Variabilität erweitert. Ein Beispiel dafür sind *Featured Transistion Systems (FTS)*, welche die bereits eingeführten Transitionssysteme um Konstrukte zur Berücksichtigung der Variabilität erweitern und in der Arbeit von Classen et al. [8] vorgestellt wurden.

2.2.2. Type Checking

> "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kind of values they compute." [12]

Type Checking ist eine statische Verifikationstechnik zur Überprüfung der Konformität eines Software-Programms bezüglich des Typsystems der zur Implementierung verwendeten Programmier-

sprache. Ein Typsystem stellt somit ein syntaktisches Konstrukt dar, um ein Programm auf die Abwesenheit von bestimmten Fehlern und Fehlverhalten zu überprüfen. Dadurch kann statisch festgestellt werden, dass diese Programmfehler nicht zur Laufzeit auftreten können [4]. Diese Überprüfung verwendet dazu Ausdrücke, welche die Werte der zur Ausführung des Programms berechneten Werte klassifizieren [12]. Ein Typsystem beschreibt dazu eine Typennotation, Sprachkonstrukte und Regeln für die Zuweisung der definierten Typen zu den Sprachkonstrukten. Die ersten Typsysteme wurden in den 1950er-Jahren für Programmiersprachen wie Fortran mit dem Ziel einer Effizienzsteigerung entwickelt. Als Beispiel konnte zwischen ganzen Zahlen und Gleitkommazahlen für arithmetische Operationen unterschieden werden, um die jeweiligen passenden effizienteren Maschinenanweisungen zu nutzen [12].
Typsysteme bieten somit eine Möglichkeit, Programme daraufhin zu überprüfen ob sie wohl-getypt sind. Beispielhaft kann mit Typsystemen untersucht werden, ob eine aufgerufene Methode verfügbar ist. Des Weiteren kann mittels eines Typsystems geprüft werden, ob die Argumente für mathematische Funktionen wie Addition oder Division immer Zahlen sind. Dahingegen kann mittels Type Checking nicht kontrolliert werden, ob das zweite Argument (*Divisor*) bei einer Division nicht die Zahl Null ist [12]. Außerdem können Typsysteme beispielsweise dazu verwendet werden, High-Level Eigenschaften bezüglich der Modularität von Programmen zu überprüfen. Exemplarisch kann in Bezug auf Kapselung von Daten (*Information Hiding*) untersucht werden, dass kein direkter Zugriff auf abstrakte Datenfelder existiert [12].
Ein Type Checker ist das Werkzeug, dass ein Programm auf die Einhaltung der Regeln des Typsystems überprüft. Type Checker werden üblicherweise in den Compiler oder den Linker der Programmiersprache integriert und während der Kompilierung oder des Linkens automatisch ausgeführt, ohne weitere Interaktionen durch den Anwender zu erfordern. Dabei wird jeder Quellprogrammkomponente durch den Compiler ein Typausdruck zugewiesen [4].
Type Checking kann in die Gruppen *Synthese* und *Inferenz* unterteilt werden [4]. Bei der Typsynthese werden die Typen der Teilausdrücke eines Ausdrucks dazu verwendet, um aus diesen Typen den Typ des Ausdrucks abzuleiten. Dazu muss ein verwendeter Name vor seiner Verwendung im Programm deklariert werden, wie beispielsweise bei der Programmiersprache *Java*.

> "wenn f den Typ s hat $\rightarrow$ (t und x haben den Typ s
> dann hat Ausdruck $f(x)$ den Typ t" [4]

Dieses Beispiel zeigt die Herleitung des Typs eines Ausdrucks $f(x)$, wobei t und x Ausdrücke sind und $s \rightarrow t$ eine Funktion darstellt.
Im Gegensatz zur Typsynthese wird bei der Typinferenz der Typ eines Ausdrucks anhand der Verwendungsweise des Ausdrucks festgelegt und deswegen müssen die Namen nicht vor der Verwendung deklariert werden [4]. Die nachfolgende Inferenzregel stellt ein Beispiel für die Typinferenz dar.

> "wenn $f(x)$ ein Ausdruck ist,
> dann hat f für ein beliebiges α und β den Typ $\alpha \rightarrow \beta$ und x hat den Typ α"
> [4]

Nachfolgend sind einige Eigenschaften und Vorteile des Type Checkings aufgelistet.

- Statisches Type Checking ermöglicht die frühzeitige Erkennung bestimmter Programmierfehler, ohne das Programm ausführen zu müssen. Durch die frühzeitige Erkennung können Fehler mit geringem Kostenaufwand behoben werden [12].
- Type Checking erlaubt eine genaue Lokalisierung der Fehlerursachen [12].
- Entdeckung von Fehlern, die sonst erst zur Laufzeit und möglicherweise nur nach einer bestimmten Reihenfolge von Funktionen des Programms auftreten würden [12].
- Type Checking kann nicht nur zur Verifikation verwendet werden, sondern auch als Werkzeug zur Wartung durch den Programmierer. Beispielsweise können nicht angepasste Aufrufe einer Methode, die zuvor umbenannt wurde, lokalisiert werden [12].

2.2.3. Theorem Proving

Theorem Proving ist eine statische Verifikationstechnik, bei der auf Grundlage eines formalen Systems und dazugehörigen Regeln versucht wird, einen Beweis für untersuchte Eigenschaften zu ermitteln [39]. Ein formales System besteht dabei aus einer formalen Sprache, Axiomen dieser Sprache sowie Inferenzregeln und weiteren Möglichkeiten zur Herleitung anderer Aussagen, welche als Theoreme bezeichnet werden [39]. Das formale System wird üblicherweise auch Kalkül genannt. Die Menge der Axiome und Inferenzregeln bilden ein sogenanntes deduktives System, die Axiome inklusive aller ableitbarer Theoreme werden als Theorie bezeichnet. Die Durchführung eines Beweises mittels Theorem Proving besteht aus einer Menge syntaktischer Umformungen der zu beweisenden Eigenschaft, die auf den Inferenzregeln des Kalküls basieren [39]. Die zu beweisende Eigenschaft wird typischerweise mittels einer prädikatenlogische Formel spezifiziert.
Ein Theorem Prover ist somit ein Werkzeug, welches den Algorithmus für das Theorem Proving implementiert und zur Herleitung von Beweisen verwendet wird. Für ein vorgegebenes Kalkül einer bestimmten Logik und einer zu beweisenden Eigenschaft in Form einer logischen Formel wird durch wiederholtes Anwenden der Inferenzregeln versucht, ein Beweis zu finden [39]. Theorem Prover unterscheiden sich unter anderem in der Möglichkeit zur Verwendung und Anpassbarkeit an benutzerdefinierte Logiken und der Automatisierung des Beweisprozesses [39]. Im Allgemeinen werden die Theorem Prover in *interaktive Theorem Prover* und *automatisierte Theorem Prover* unterteilt.

> "Interactive theorem provers (ITPs) are systems which allow to process logical formulas and to apply inference rules upon them." [39]

Interaktive Theorem Prover benötigen Interaktionen des Anwenders, um schrittweise einen Beweis durch die Anwendung von Inferenzregeln in Verbindung mit vordefinierten Aktionen in Form von Anweisungsskripten durchzuführen [39].

> "Automated theorem provers are programs which, given a formula, try to evaluate whether the formula is universally valid or not." [39]

Automatisierte Theorem Prover bieten einen vollautomatisierten Beweis der allgemeinen Gültigkeit einer logischen Formel [39].

Automatische Theorem Prover können zusätzlich in *Resolutions-basierte Theorem Prover* und *Theorem Prover basierend auf Kalkülen mit anschaulichen Darstellungen* (engl. *tableau-based calculi*) unterteilt werden [39]. Resolutions-basierte Theorem Prover beginnen den Beweisprozess mit den vorhandenen Axiomen und leiten daraus neue Formeln ab [39]. Theorem Prover mit Tableau-basierten Kalkülen generieren für die Durchführung des Beweises eine spezifische Datenstruktur, die alle Teile der aufgebrochenen zu beweisenden Formel enthält. Wenn diese Datenstruktur bestimmte Eigenschaften aufweist, ist die zu beweisende Formel nicht erfüllbar [39]. Beide dieser Formen automatisierter Theorem Prover beweisen eine Formel nicht direkt, sondern indirekt durch Beweisen der Nicht-Erfüllbarkeit der Negation der Formel [39].
Automatische Theorem Prover akzeptieren üblicherweise nur logische Formeln in Prädikatenlogik 1. Ordnung in konjunktiver Normalform. Prädikatenlogik erweitert die Aussagenlogik um den Existenzquantor ($\exists$) und den Allquantor ($\forall$) [39]. Mit diesen Quantoren können Aussagen über alle Variablen beziehungsweise mindestens eine Variable einer Formel getroffen werden. Eine ausführliche Einführung dieser Logik ist in dem Buch von Johann Schumann [39] zu finden. Die konjunktive Normalform ist eine Konjunktion von Klauseln, wobei eine Klausel eine Disjunktion atomarer Formeln oder deren Negation darstellt [39]. Die folgende Formel zeigt ein Beispiel für eine Formel in konjunktiver Normalform, wobei A, B, C und D atomare Formeln repräsentieren.

$$(A \vee B) \wedge C \wedge \neg D$$

Interaktive und automatische Theorem Prover unterscheiden sich in einigen Eigenschaften. Interaktive Theorem Prover sind ausdrucksmächtig und können eine Vielzahl an Logiken wie maßgeschneiderte Logik, Logiken höherer Ordnungen sowie getypte Logiken unterstützen. Automatische Theorem Prover hingegen erlauben hauptsächlich die Verwendung von einer Prädikatenlogik 1. Ordnung, da diese Logiken effektive Inferenzregeln und Beweisprozeduren zur automatisierten Beweisdurchführung bieten [39]. Des Weiteren besitzen interaktive Theorem Prover häufig Features wie die automatische Wiederholung eines bereits durchgeführten Beweises oder das Hinzufügen von Beweis-Theoremen zu einer Datenbank für eine spätere Wiederverwendung [39]. Außerdem ermöglichen interaktive Theorem Prover eine intuitive Formulierung der Beweisaufgaben. Des Weiteren existieren oftmals Techniken zur Integration von komplexeren Operationen wie beispielsweise Induktion in das darunterliegende Logik-Framework. Somit können auch umfangreichere Beweise realisiert werden [39]. Ein Nachteil von interaktiven Theorem Provern ist der Bedarf an umfangreicher Anwenderinteraktion, sodass die Beweise oftmals sehr lange dauern. Außerdem ist detailliertes Wissen bezüglich der Anwendungsdomäne, der verwendeten Logik sowie des Theorem Provers notwendig, wodurch die Beweise bei interaktiven Theorem Provern im Allgemeinen nur von Spezialisten durchgeführt werden können [39]. Automatisierte Theorem Prover weisen häufig ebenfalls lange Laufzeiten zum Finden eines Beweises auf. Zusätzlich besteht die Möglichkeit, dass beim automatisierten Theorem Proving trotz langer Beweissuche kein Beweis gefunden wird [39]. Weiterhin sind mittels automatisiertem Theorem Proving keine komplexen Beweise ohne zusätzliche Anwenderinteraktion möglich. Ein weiterer Nachteil automatischer Theorem Prover ist typischerweise der Mangel einer Benutzerschnittstelle, sodass der Theorem Prover ausschließlich aus Suchalgorithmen besteht und keine Rückmeldung bietet, wenn ein Beweis nicht gefunden werden konnte [39].

2.2.4. Static Analysis

"Any tool that analyzes code without executing it is performing static analysis" [14]

Der Begriff der statischen Analyse umfasst somit eine Vielzahl von Programmanalysen, die das analysierte Programm dabei nicht ausführen. Wie bei allen statischen Analysen wird ausschließlich sichergestellt, dass bestimmte Fehlerarten im Programm nicht vorhanden sind. Es kann jedoch nicht analysiert werden, ob ein komplett fehlerfreies, perfektes Programm vorliegt [14]. Wie beschrieben, enthält die Klasse der statischen Analyse verschiedene konkrete Analysetechniken. Der Begriff *Static Analysis* umfasst in der vorliegenden Arbeit hauptsächlich die Datenflussanalyse. Dementsprechend wird nachfolgend die Datenflussanalyse als Beispiel für eine statische Analyse beschrieben, da die Datenflussanalyse die vorwiegende Analysetechnik in den Primärstudien zu Static Analysis darstellt.
Bei der Datenflussanalyse werden Informationen über den Datenfluss entlang der Ausführungspfade eines Programms ermittelt [4]. Beispielsweise kann damit eine *tote Wertezuweisung* ermittelt werden. Eine tote Wertezuweisung ist eine Zuweisung, dessen Ergebnis in keinem der nachfolgend möglichen Ausführungspfade verwendet wird und daher aus dem Programm entfernt werden kann [4]. Die Datenflussanalyse besteht normalerweise aus einem Kontrollflussgraphen, einem Gitternetzwerk (engl. *lattice*) und einer Menge von Übertragungsfunktionen [23]. Der Kontrollflussgraph wird für die eigentliche Analyseprozedur verwendet, das Gitternetzwerk enthält die relevanten Werte der Analyse und die Übertragungsfunktionen simulieren das Verhalten des Programms zur Ausführungszeit [23].
Der Kontrollflussgraph ist ein gerichteter Graph, dessen Knoten die Anweisungen des Quellprogramms und die Kanten den Kontrollfluss des Programms repräsentieren [23]. Des Weiteren werden die Zustände vor und hinter einem Knoten mit Programmpunkten annotiert. Ein Programmpunkt bezeichnet üblicherweise den Zustand des Programms vor beziehungsweise nach der Ausführung des Programms. Ein Programmzustand besteht dabei aus den Variablenwerten des Programms [4].
Das Gitternetzwerk enthält die berechneten Werte, die während der Datenflussanalyse berechnet werden. Ein Gitternetzwerk wird formal durch $\mathcal{L} = (D, \subseteq)$ definiert, wobei D die Menge der Elemente darstellt und $\subseteq$ eine partielle Ordnung bezeichnet [23].
Die Simulation des Verhaltens zur Kompilierungszeit wird durch eine Menge von Übertragungsfunktionen beschrieben, welche für jede Anweisung im Programm eine Übertragungsfunktion der Form $f_S : \mathcal{L} \rightarrow \mathcal{L}$ enthält [23].

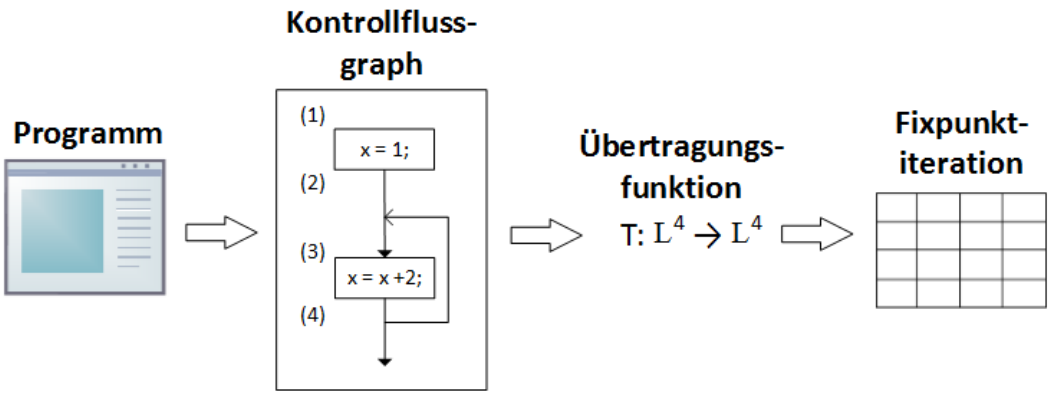


Abbildung 2.6.: Datenflussanalyse - Vereinfachte Darstellung einer möglichen Analyseprozedur (orientiert an [23])

Die Analyseprozedur der Datenflussanalyse wird durch die Abbildung 2.6 illustriert. Ein Programm wird im ersten Schritt in einen Kontrollflussgraphen überführt. In dem Beispiel besteht das Programm aus zwei Anweisungen, der Zuweisung des Wertes 1 an die Variable *x* sowie der Addition der Variablen *x* um den Zahlenwert 2. Somit existieren in diesem Kontrollflussgraphen insgesamt die vier Programmpunkte *(1), (2), (3) und (4),* jeweils vor und nach einer Anweisung. Im nächsten Schritt werden die Übertragungsfunktionen für jeden Programmpunkt erstellt, sodass die Programmübertragungsfunktion aus insgesamt vier Funktionen besteht. Diese vier Funktionen bilden zusammen betrachtet ein Gleichungssystem. Im letzten Schritt wird mit Hilfe einer Fixpunktiteration eine Näherungslösung für das vorliegende Gleichungssystem, bestehend aus den Übertragungsfunktionen gesucht, welche z.B. in tabellarischer Form dargestellt werden kann. Der Begriff der Datenflussanalyse umfasst ebenfalls eine Menge von konkreteren Analysen. Dazu zählen die Analysemethoden *erreichende Definitionen* (engl. *reaching definitions*), *Analyse lebendiger Variablen*) sowie *verfügbare Variablen* [4].

Bei der Analyse der *erreichenden Definitionen* werden Informationen anhand von Definitionen von Variablen an bestimmten Programmpunkten ermittelt [4]. Eine Definition stellt eine Anweisung zur Wertzuweisung beziehungsweise einer möglichen Wertzuweisung dar [4]. Wenn ein direkter Pfad vom nachfolgenden Programmpunkt einer Definition zu einem spezifischen späteren Programmpunkt existiert, erreicht diese Definition diesen spezifischen Programmpunkt. Erreicht eine Definition einen Programmpunkt, wird sie auf dem Pfad zwischen der Definition und dem Programmpunkt nicht zerstört [4]. Die Zerstörung einer Definition einer Variablen wird durch eine erneute spätere Definition dieser Variablen an einem späteren Programmpunkt erreicht. Somit besagt eine erreichende Definition einer Variablen, dass der Wert dieser Variablen das letzte Mal an der Stelle dieser Definition zugewiesen wurde.

Eine Beispiel der erreichenden Definitionen ist die Überprüfung durch den Compiler, ob eine Variable an einem spezifischen Programmpunkt eine Konstante ist [4]. Ein Debugger kann mittels erreichbaren Definitionen untersuchen, ob eine Variable an einem spezifischen Programmpunkt, an dem die Variable verwendet wird, nicht definiert ist.

Die Analyse lebendiger Variablen überprüft, ob der Wert einer Variablen an einem spezifischen Programmpunkt entlang eines Pfades des Kontrollflussgraphen, der in diesem Programmpunkt beginnt, verwendet werden kann [4]. Das Ergebnis der Analyse lebendiger Variablen ist die Einstufung in für diesen Programmpunkt lebendige beziehungsweise tote Variablen.
Die Analyse lebendiger Variablen kann beispielsweise bei der Registervergabe für Grundblöcke hilfreich sein [4]. Als Grundblock werden diejenigen Teile bezeichnet, bei denen der Programmpunkt hinter einer Anweisung derselbe wie der Programmpunkt vor der nachfolgenden Anweisung ist [4]. Mittels der Analyse lebendiger Variablen kann ermittelt werden, ob ein in einem Register berechneter Wert am Ende eines Blockes tot ist und somit nicht gespeichert werden muss. Um dieses Ergebnis zu nutzen, sollten Register mit toten Werten bei der Auswahl eines neuen Registers präferiert werden, wenn das aktuelle Register voll ist [4].

Die Untersuchung der verfügbaren Ausdrücke ist eine weitere Anwendung der Datenflussanalyse. Dabei wird überprüft, ob ein Ausdruck an einem Programmpunkt verfügbar ist. Ein Ausdruck ist in einem Programmpunkt P_a verfügbar, wenn der Ausdruck in jedem Programmpunkt vor P_a ausgewertet wird und dabei für keine Komponente einer Ausweisung eine Zuweisung erfolgt [4]. Ein Ausdruck ist beispielsweise die Konkatenation der zwei Zeichenketten $String_A + String_B$. Bei der Untersuchung der verfügbaren Ausdrücke wird ein Ausdruck zerstört, wenn eine Komponente eines Ausdrucks einen Wert zugewiesen bekommt und dadurch nicht neu berechnet wird [4]. Dahingegen wird ein Ausdruck von einem Block generiert, wenn der Block den Ausdruck auswertet und kein Teil des Ausdruckes definiert wird [4].
Die Analyse der verfügbaren Ausdrücke kann beispielsweise zur Ermittlung gemeinsamer globaler Teilausdrücke genutzt werden [4]. Ein globaler gemeinsamer Teilausdruck ist ein Teilausdruck, der bereits berechnet wurde und seit dieser Berechnung in keinem der möglichen nachfolgenden Blöcke entlang aller möglichen Pfade verändert wurde [4].

3 Verifikation von Software-Produktlinien

Wird die Tatsache betrachtet, dass mittels Software-Produktlinien mehrere unterschiedliche Produkte erzeugt werden können, impliziert die Verifikation aller dieser Produkte einen erhöhten Verifikationsaufwand gegenüber Einzelsystemen und Standardsoftware. Die Produkte einer Software-Produktlinie besitzen oftmals viele Gemeinsamkeiten durch die wiederverwendbaren Implementierungsartefakte, wodurch bei der separaten Verifikation der Produkte viele redundante Berechnungen durchgeführt werden [79]. Jedoch bieten Software-Produktlinien durch Verwendung dieser Gemeinsamkeiten und verschiedener Verifikationsstrategien eine Möglichkeit, den Verifikationsaufwand zu verringern [79].

Da eine Software-Produktlinie mit n Features bis zu 2^n Produkte generieren kann, ist eine separate Verifikation jedes einzelnen Produkts aufgrund der benötigten Zeit und des benötigten Aufwands oftmals unmöglich [79]. Eine parallele Verifikation aller Produkte erfordert beispielsweise einen hohen Aufwand in Form von benötigtem Arbeitsspeicher und ist oftmals aufgrund des verfügbaren Arbeitsspeichers unmöglich. Eine Software-Produktlinie mit 33 optionalen Features ohne weitere Cross-Tree-Constraints erzeugt mehr Produkte als es Menschen auf der Erde gibt. Beispielsweise würde die einzelne Verifikation all dieser Produkte mehr als 272 Jahre dauern, wenn die Verifikation jedes Produkts nur eine Sekunde dauern würde [79]. Üblicherweise dauert die Verifikation eines Produkts deutlich länger als eine Sekunde und es existieren Software-Produktlinien mit deutlich mehr als 33 Features. Diese Tatsachen verdeutlichen, dass bei einer Vielzahl von Software-Produktlinien die separate Verifikation aller Produkte nicht möglich ist. Der hohe Verifikationsaufwand ist allerdings nicht nur ein Problem für große Software-Produktlinien, sondern auch kleinere Produktlinien mit wenig Features können einen deutlichen Performanzunterschied durch die jeweiligen Verifikationsstrategien verzeichnen.

Im folgenden Kapitel 3.1 werden die verschiedenen Verifikationsstrategien vorgestellt, die von Thüm et al. [79] identifiziert wurden. In Kapitel 3.2 wird die Problemstellung dieser Arbeit ausführlich erläutert. Kapitel 3.3 stellt den Prozess zur systematischen Literaturrecherche der wissenschaftlichen Dokumente zu den Techniken zur statischen Analyse von Software-Produktlinien vor und gibt einen Überblick über die Unterschiede und Gemeinsamkeiten der verschiedenen Analysetechniken. Im Abschnitt 3.4 werden zum einen die in den untersuchten Dokumenten enthaltenen Hypothesen und Behauptungen bezüglich der Performanz der verschiedenen Strategien vorgestellt. Zum anderen werden im Abschnitt 3.4 die daraus abgeleiteten Hypothesen erläutert, die in der vorliegenden Arbeit untersucht werden.

3.1. Verifikationsstrategien für Software-Produktlinien

Da es oftmals unmöglich ist alle Produkte einer Software-Produktlinie einzeln zu verifizieren, wurden in den letzten Jahren viele verschiedene Strategien entwickelt, die die Variabilität und Gemeinsamkeiten von Software-Produktlinien nutzen, um den Aufwand für die Verifikation zu verringern [79]. Der Verifikationsaufwand bezieht sich sowohl auf die benötigte Zeit als auch auf den benötigten Arbeitsspeicher. Thüm et al. [79] haben 123 wissenschaftliche Artikel zur Verifikation von Software-Produktlinien untersucht und die Verifikationsansätze dabei in unterschiedliche Strategien eingeordnet. Thüm et al. [79] haben dabei drei Hauptstrategien ermittelt. Dazu zählen die *Produkt-basierte Strategie*, die *Familien-basierte Strategie* und die *Feature-basierte Strategie*. Die Ansätze wurden danach klassifiziert, auf welche Art und Weise sie den Analyseaufwand versuchen zu verringern. Des Weiteren wurden dabei Kombinationen dieser Strategien entwickelt. Diese Strategien haben unterschiedliche Vor- und Nachteile, weshalb nicht jede Strategie für jede Software-Produktlinie gleich gut geeignet ist. Teilweise müssen existierende Analysetechniken aus der traditionellen Software-Verifikation angepasst werden, um diese unterschiedlichen Strategien unterstützen zu können. Die ermittelten Verifikationsstrategien werden in den folgenden Unterkapiteln vorgestellt und ihre Vor- und Nachteile werden aufgelistet.

3.1.1. Produkt-basierte Strategie

> "An analysis of a software product line is product based if it operates only on generated products or models thereof, whereas the variability model may be used to generate all products or to implement optimizations. A product-based analysis is called optimized if it operates on a subset of all products (a.k.a. sample-based analysis) or if intermediate analysis results of some products are reused for other products; it is called unoptimized otherwise (a.k.a. exhaustive, comprehensive, brute-force, and feature-oblivious analysis)." [79]

Diese Definition wurde von Thüm et al. [79] in der Arbeit *A Classification and Survey of Analysis Strategies for Software Product Lines* formuliert. Sie beschreiben die Produkt-basierte Strategie als eine Methode, die sich durch Generierung aller Produkte und separater Analyse dieser Produkte mit existierenden Model-Checking-Techniken und Werkzeugen auszeichnet [79]. Eine simple Form dieser Strategie die Generierung jedes Produkt und die anschließende iterative Analyse jedes dieser Produkte. Dieser *Produkt-basierte Ansatz* bildet oftmals die Basis für den Performanz-Vergleich der Ansätze, die auf anderen Verifikationsstrategien basieren. Jedoch wird oftmals ein sogenannter *Sample-basierter Ansatz* verwendet, wobei nur eine Teilmenge aller möglichen Produkte der Software-Produktlinie analysiert wird [79]. Hierbei werden verschiedene Abdeckungskriterien genutzt, mit denen bestimmte Korrektheitsergebnisse mit einer hohen Wahrscheinlichkeit für die gesamte Software-Produktlinie angenommen werden können [79]. Außerdem unterteilen Thüm et al. [79] die Produkt-basierte Strategie in die *optimierte Produkt-basierte Strategie* und die *unoptimierte Produkt-basierte Strategie*. Die unoptimierte Produkt-basierte Strategie repräsentiert den Ansatz, bei dem alle Produkte generiert, separat analysiert und keine Ergebnisse dieser Analysen zur Verifikation der anderen Produkte wiederverwendet werden. Die optimierte Produkt-basierte Strategie umfasst zum einen Sample-basierte Ansätze. Zum anderen umfasst sie auch Strategien, die alle oder eine Teilmenge der Produkte generieren, verifizieren und bei der Verifikation der weiteren

Produkte die Analyseergebnisse von vorherigen Produkten wiederverwenden [79].

Beispiel Im fortlaufenden Beispiel des Mp3-Players (siehe Abbildung 2.1) können die vier möglichen Produkte *TimeProgress*, *PercentProgress*, *TimeProgress + RestartSong* und *PercentProgress + RestartSong* generiert werden. Diese Produkte der Software-Produktlinie könnten auf folgende Eigenschaft überprüft werden, dass immer wenn der Mp3-Player einen Musiktitel abspielt irgendwann die Fortschrittsanzeige aktualisiert wird. Diese Eigenschaft wird durch folgende temporallogische Formel spezifiziert.

$$\Box playing \Rightarrow \Diamond updatingProgress$$

Wie in Abbildung 3.1 dargestellt, muss bei der Produkt-basierten Strategie für jede mögliche valide Feature-Auswahl im ersten Schritt das entsprechende Produkt generiert werden. Die Feature-Auswahl ist in dieser Abbildung mit *Konf. x* annotiert, das Software-Produkt mit *Prog. x* bezeichnet. In der gesamten Abbildung kann *x* die Werte 1, 2, 3 oder 4 annehmen. Das Verhalten der generierten Produkte muss anschließend für das Model Checking mittels Transitionssystemen modelliert werden, welche in der Abbildung mit *TS x* abgekürzt werden. Jedes dieser Transitionssysteme bildet zusammen mit der logischen Formel, welche die überprüfte Eigenschaft darstellt, die Eingabe des Model Checkers *MC*. Die beispielhafte Eigenschaft wird von keinem Produkt verletzt und die Einhaltung der spezifizierten Eigenschaft wird in der Abbildung 3.1 durch eine ausgewählte Checkbox dargestellt. Das Beispiel der Produkt-basierten Strategie in Abbildung 3.1 ist eine vereinfachte Darstellung und erhebt keinen Anspruch auf Vollständigkeit.

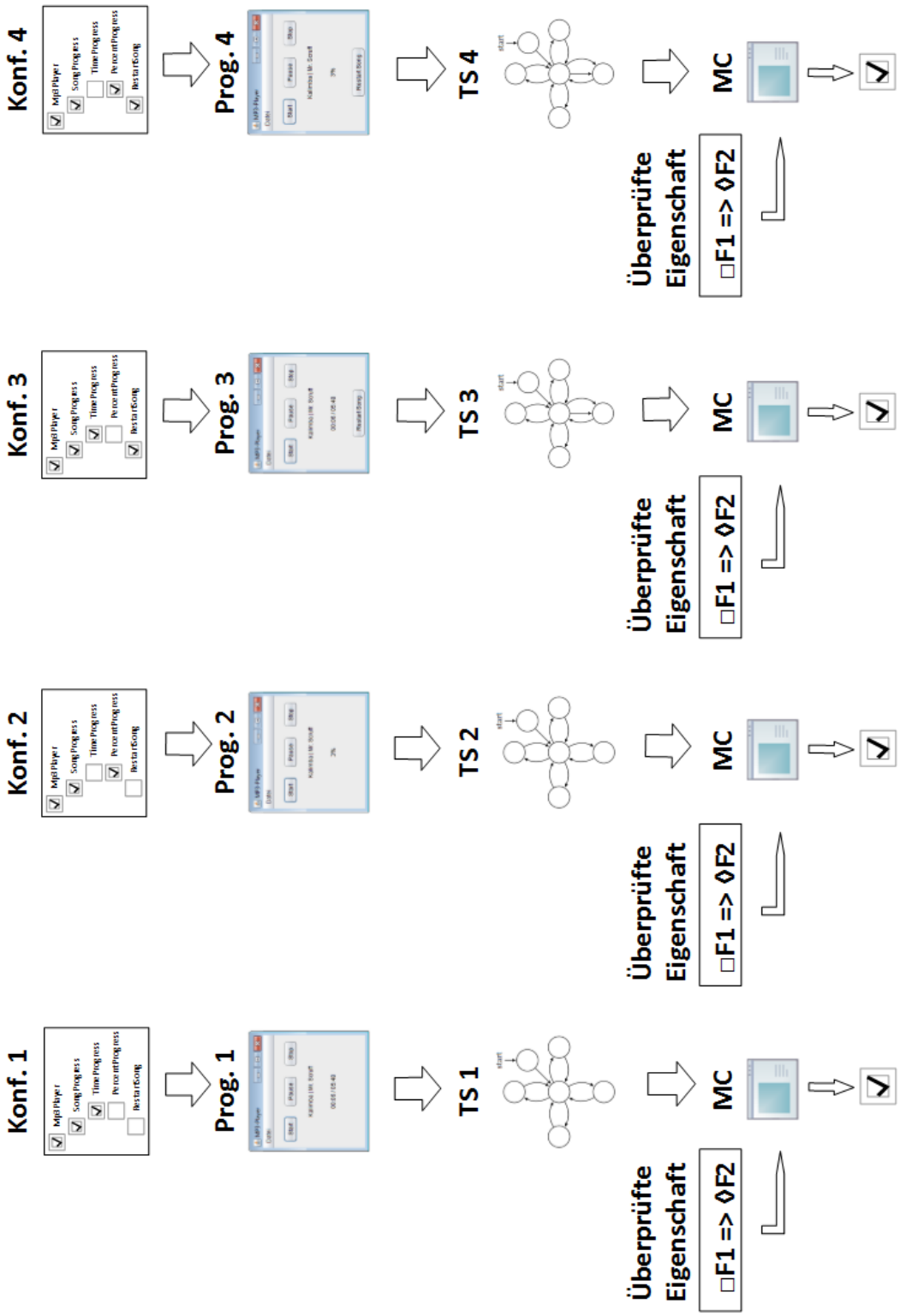


Abbildung 3.1.: Beispiel der Produkt-basierten Strategie für das fortlaufende Mp3-Player-Beispiel

Unoptimierte Produkt-basierte Strategie

Wie im vorherigen Kapitel beschrieben, bildet die Performanz der unoptimierten Produkt-basierten Strategie oftmals die Basis für die Evaluierungen der neuen Verifikationsansätze. Denn die unoptimierte Produkt-basierte Strategie führt ohne weitere Optimierungsmethoden zu keiner Reduzierung des Verifikationsaufwands, da sie alle Produkte generiert und verifiziert. Hierbei ist zu erwähnen, dass die Generierung der Produkte alleine oft schon viel Zeit in Anspruch nimmt und bei einer großen Anzahl an Produkten sogar nicht durchführbar ist [79]. Ist die Generierung aller Produkte einer Software-Produktlinie möglich, ist die einzelne Analyse aller generierten Produkte aufgrund von zu hohem Aufwand häufig unmöglich. Da die Produkte einer Software-Produktlinie oftmals viele Gemeinsamkeiten haben und sich nur in wenigen Features unterscheiden, entstehen bei der iterativen Produkt-basierten Analyse redundante Berechnungen [79].

Optimierte Produkt-basierte Strategie

Die optimierte Produkt-basierte Strategie hingegen reduziert den Aufwand der Verifikation auf zwei verschiedene Arten. Entweder untersuchen die Ansätze dieser Strategieklasse die Produkte auf redundante Teile, die in mehreren Produkten vorkommen und verifizieren diese nur einmal. Das Analyseergebnis dieser redundanten Teile wird anschließend während der Analyse der Produkte, die diese Teile ebenfalls enthalten, wiederverwendet [79]. Oder die Ansätze identifizieren Produkte, deren Verifikation andere Produkte mit abdecken und somit nur eine Teilmenge aller möglichen Produkte verifiziert [79].

3.1.2. Familien-basierte Strategie

> "An analysis of a software product line is family based if it (a) operates only on domain artifacts and (b) incorporates the knowledge about valid feature combinations." [79]

Laut der Definition von Thüm et al. [79] operiert die Familien-basierte Strategie anders als die Produkt-basierte Strategie auf den Domänenartefakten anstatt auf den Produkten. Zusätzlich werden alle validen Kombinationen der Domänenartefakte berücksichtigt. Dazu wird das Variabilitätsmodell, üblicherweise das Feature-Modell, in eine logische Formel umgewandelt, um alle validen Feature-Kombinationen während der Verifikation zu analysieren [79]. Dadurch kann beispielsweise mittels eines SAT-Solvers überprüft werden, ob eine Methode in allen Feature-Kombinationen verfügbar ist, in denen sie verwendet wird [79]. Die Familien-basierte Analyse kann entweder durch die Erzeugung eines Metaprodukts, durch Adaptierung eines existierenden Model Checkers oder durch eine Kombination der beiden ersten Formen umgesetzt werden. Ein Metaprodukt ist ein virtuelles Produkt, welches alle möglichen Feature-Kombinationen beinhaltet [79]. Das Metaprodukt muss nicht zwingend ein valides Produkt gemäß des Variabilitätsmodells sein, da optionale und alternative Features in Kombinationen enthalten sein können, die gemäß des Feature-Modells nicht valide sind. Die Familien-basierte Strategie verfolgt den Ansatz, durch Verifikation der Domänenartefakte sowie deren Kombinationen als auch des Variabilitätsmodells die Eigenschaften für alle Produkte einer Software-Produktlinie zu verifizieren [79].

Beispiel Die Familien-basierte Strategie kann dazu verwendet werden, alle möglichen Feature-Kombinationen der Software-Produktlinie des fortlaufenden Beispiels des Mp3-Players in einem einzelnen Durchlauf zu verifizieren. Beispielsweise könnte die spezifizierte Eigenschaft aus dem

Beispiel der Produkt-basierten Strategie verifiziert werden. Dafür könnten beispielsweise alle möglichen Produkte wie in Abbildung 3.2 durch ein einziges FTS modelliert werden. Alle möglichen Produkte sind in der Abbildung 3.2 durch das Feature-Diagramm dargestellt. Das FTS besitzt bei der Aktion *pushRestart*, getrennt durch einen Schrägstrich, das Feature *RestartSong*, welches diese Kante bereitstellt. Dabei ist zu erwähnen, dass das beispielhafte FTS keinen Anspruch auf Validität erhebt, sondern nur als Beispiel für eine optionale Verbindung im FTS für ein optionales Feature dient. Das FTS bildet zusammen mit der temporallogischen Formel die Eingabe des Model Checkers, welcher in einem Durchlauf alle möglichen Produkte gegen diese Eigenschaft überprüft. Das Ergebnis des Model Checkings ist dabei eine Formel, die alle Produkte aufzeigt, welche die Eigenschaft verletzten. Die beispielhafte Eigenschaft wird von keinem Produkt verletzt, weshalb das Ergebnis des Model Checkers in der Abbildung 3.2 leer dargestellt wird.

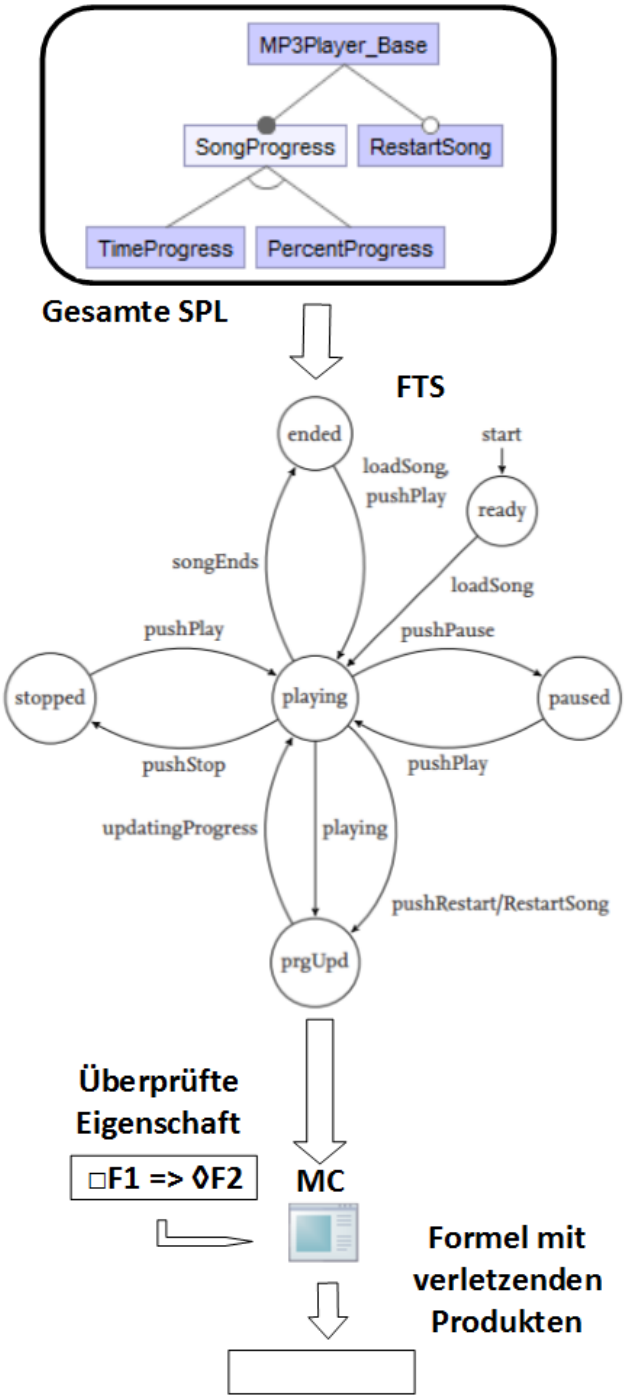


Abbildung 3.2.: Beispiel der Familien-basierten Strategie für das fortlaufende Mp3-Player-Beispiel

3.1.3. Feature-basierte Strategie

Eine weitere Verifikationsstrategie aus der Arbeit von Thüm et al. [79] ist die Feature-basierte Strategie.

> "An analysis of a software product line is feature based if (a) it operates only on domain

> artifacts and (b) software artifacts belonging to a feature are analyzed in isolation (i.e., knowledge about valid feature combinations is not used) and feature interactions are not considered.“ [79]

Diese Definition entstammt der Arbeit Thüm et al. [79] und besagt, dass durch die Feature-basierte Strategie ebenfalls Domänenartefakte anstatt generierter Produkte verifiziert werden [79]. Dabei werden die Domänenartefakte isoliert verifiziert, die ein einzelnes Feature implementieren. Dazu muss das Variabilitätsmodell nicht verfügbar sein, da bei der Feature-basierten Strategie keine Feature-Kombinationen betrachtet werden [79]. Durch die Feature-basierte Analyse verringert sich der Analyseaufwand von einem exponentiellen Aufwand für die Verifikation aller möglichen Feature-Kombinationen auf einen linearen Aufwand entsprechend der Anzahl der Features. Die Feature-basierte Strategie verwendet die Annahme, dass einige Eigenschaften durch die isolierte Verifikation einzelner Features analysierbar sind [79]. Die Feature-basierte Strategie kann dagegen unvollständig sein, da es keine Feature-Kombinationen berücksichtigt und somit keine Eigenschaften verifizieren kann, die sich auf mehrere Features beziehen [79]. Dadurch können Probleme wie das *Feature-Interaktionsproblem* aus dem Kapitel 2.1 nicht erkannt werden.

Beispiel Abbildung 3.3 zeigt die Feature-basierte Strategie für das fortlaufende Beispiels des Mp3-Players. Wie in der Abbildung dargestellt, werden alle konkreten Features in eigenständige Teile von Transitionssystemen übersetzt, welche mit *TS_Y* annotiert sind, wobei *Y* als Variable für das jeweilige Feature dient. Anschließend würde jedes dieser Transitionssysteme eines Features zusammen mit der Eigenschaftsspezifikation in Form einer temporallogischen Formel vom Model Checker verifiziert werden. Dabei wird als Eigenschaft die im Beispiel der Produkt-basierten Strategie spezifizierte Eigenschaft verwendet. Das Teil-Transitionssystem *TS_Progress* wird einmal für das Feature *TimeProgress* und einmal für das Feature *PercentProgress* verifiziert. Die Transitionssysteme TS_Timeprogress und TS_Percentprogress verletzen die spezifizierte Eigenschaft nicht. Dieses Ergebnis des Model Checkers ist durch eine abgehakte Checkbox dargestellt. Für die Teiltransitionssysteme TS_Base und TS_Restart kann die Eigenschaft nicht überprüft werden, da diese Modelle die relevanten Zustände und Transitionen nicht vollständig enthalten. Dies wird in der Abbildung durch das eingekreiste Fragezeichen dargestellt. Dieses Beispiel stellt die Feature-basierte Strategie ausschließlich vereinfacht dar und garantiert daher keinen vollständig korrekten, realen Verifikationsprozess.

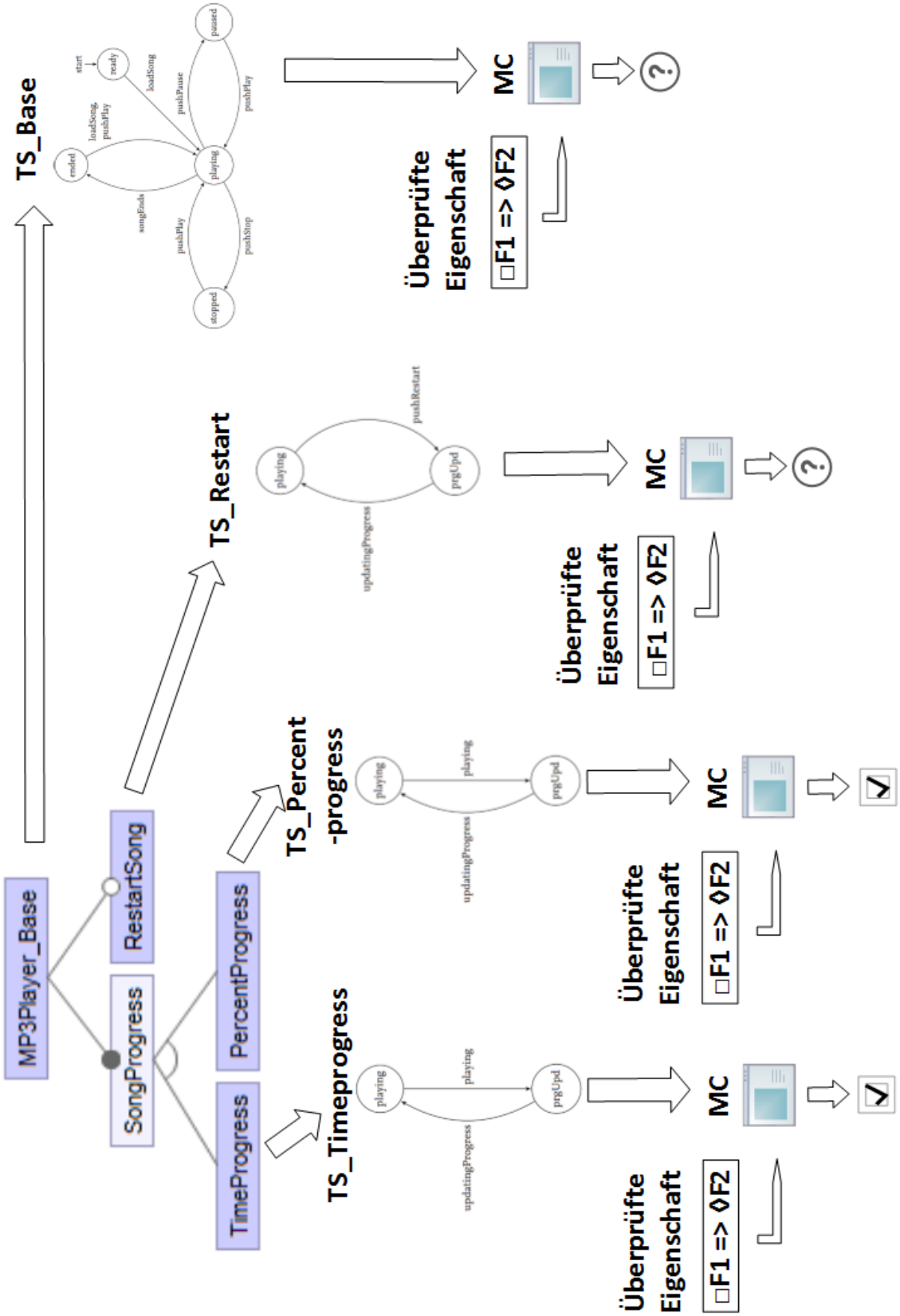


Abbildung 3.3.: Beispiel der Produkt-basierten Strategie für das fortlaufende Mp3-Player-Beispiel

3.1.4. Kombinationen der primären Verifikationsstrategien

Thüm et al. [79] haben auch Kombinationen der drei in den letzten Kapiteln beschriebenen Strategien identifiziert. Dazu gehören die folgenden Strategie-Kombinationen.

- Feature-Produkt-basierte Strategie
- Feature-Familien-basierte Strategie
- Familien-Produkt-basierte Strategie
- Feature-Familien-Produkt-basierte Strategie

Theoretisch sind alle dieser Kombinationen möglich, Thüm et al. [79] haben allerdings zur Feature-Familien-Produkt-basierten Strategie keine Ansätze in ihrer Literaturrecherche ermittelt. Zusätzlich sind einige dieser Strategien nur bei unterschiedlichen Analysetechniken präsent. Beispielsweise wurden von Thüm et al. [79] nur Feature-Produkt-basierte Strategien bezüglich des Model Checkings identifiziert [79]. Während der Literaturrecherche für diese Studie wurden keine Ansätze ermittelt, welche empirische Evaluierungen zu Kombinationen der Strategien für Model-Checking-Techniken enthielten.
Die Kombinationen der primären Strategien unterteilen die Verifikation in mehrere Teilschritte. Exemplarisch werden im ersten Schritt der Feature-Produkt-basierten Strategie alle Features isoliert mittels einer Feature-basierten Strategie verifiziert. Im zweiten Schritt werden alle Produkte separat verifiziert, da die Feature-basierte Strategie nur Eigenschaften bezüglich einzelner Features verifizieren kann. Oftmals nutzen die Kombinationsstrategien die Verifikationsergebnisse der vorherigen Teilschritte. Für Definitionen und weitere Erläuterungen der Kombinationsstrategien wird auf die Arbeit von Thüm et al. [79] verwiesen.

3.1.5. Eigenschaften und Vor- & Nachteile der Strategien

Die folgende Tabelle 3.5 stellt einen Vergleich der Eigenschaften der Produkt-basierten Strategie, der Familien-basierten Strategie und der Feature-basierten Strategie dar.

	Produkt-basierte Strategie	**Familien-basierte Strategie**	**Feature-basierte Strategie**
Was wird verifiziert?	Generierte Produkte	Domänenartefakte	Domänenartefakte
Wiederverwendung bestehender Werkzeuge?	Ja	Ja/Nein	Ja/Nein
Bei Änderungen im Variabilitätsmodell?	Nur neue Produkte müssen verifiziert werden	Komplette Software-Produktlinie muss verifiziert werden	Keine neue Verifikation notwendig
Bei Hinzufügen/Änderung von Features	Neue/geänderte Produkte müssen verifiziert werden	Komplette Software-Produktlinie muss verifiziert werden	Neue/geänderte Features müssen verifiziert werden
Redundante Berechnungen?	Ja	Nein	Nein
Annahmen?	Closed-World	Closed-World	Open-World/ Closed-World
State Space beim Model Checking	Vergleichsbasis	Größer/Kleiner als bei Produkt-basiert	Deutlich kleiner als bei Produkt-basiert

Tabelle 3.5.: Eigenschaften und Vor- & Nachteile der drei primären Verifikationsstrategien

In der Tabelle 3.5 wird gezeigt, dass die Produkt-basierte Strategie auf generierte Produkte angewendet wird, wohingegen sowohl die Familien-basierte Strategie als auch die Feature-basierte Strategie auf den Domänenartefakten agieren. Des Weiteren zeigt die Tabelle, dass theoretisch für alle drei Strategien existierende Werkzeuge für Model Checking verwendet werden können. In der Praxis ist diese Wiederverwendung für die Familien-basierte und Feature-basierte Strategie nicht immer möglich, oftmals müssen die Werkzeuge für diese Strategien angepasst werden.
Außerdem unterscheiden sich diese drei Strategien im Verhalten bei Änderungen des Variabilitätsmodells wie beispielsweise die Änderung einer *Und-Beziehung* zu einer *Oder-Beziehung* im Feature-Diagramm. Bei der Produkt-basierten Strategie müssen die durch die Änderung entstandenen Produkte verifiziert werden. Die Familien-basierte Strategie hingegen erfordert eine erneute Verifikation der kompletten Software-Produktlinie. Die Feature-basierte Strategie benötigt bei Änderungen des Variabilitätsmodells keine zusätzlichen Verifikationen, da eine solche Änderung das Verhalten der Features nicht beeinflusst. Wenn bei einer Software-Produktlinie Features hinzugefügt oder geändert werden, müssen bei der Produkt-basierten Strategie die neuen oder geänderten Produkte verifiziert werden. Bei der Familien-basierten Strategie muss die komplette Software-Produktlinie erneut verifiziert werden, wohingegen die Feature-basierte Strategie nur eine Verifikation der neuen oder geänderten Features benötigt.
Einen weiteren Unterschied zwischen diesen drei Strategien, der in der Tabelle 3.5 ersichtlich ist, bezeichnet die Ausführung von redundanten Berechnungen während der Verifikation. Die Tabelle zeigt ebenfalls, dass die Produkt-basierte und die Familien-basierte Strategie die Annahme einer

Closed-World treffen. Closed-World bedeutet, dass zur Verifikation alle Features verfügbar sind und alle Beziehungen zwischen den Features bekannt sind. Die Annahme einer Closed-World ist generell immer notwendig, wenn die komplette Software-Produktlinie verifiziert werden soll. Theoretisch ist die Produkt-basierte Strategie auch unter der Annahme einer *Open-World* möglich, in diesem Fall werden jedoch nicht alle möglichen Produkte verifiziert. Die Feature-basierte Strategie erlaubt sowohl die Verwendung der *Closed-World* als auch der *Open-World*. Durch die Annahme einer Open World können alle verfügbaren Features verifiziert werden, auch wenn die Software-Produktlinie nicht vollständig bekannt ist.
Die letzte Zeile der Tabelle 3.5 zeigt die Größe des Zustandsraums beim Model Checking der jeweiligen Strategie. Dabei dient die Produkt-basierte Strategie als Basis für den Vergleich mit der Familien-basierten Strategie und der Feature-basierten Strategie. Beim Vergleich des State Spaces der Familien-basierten Strategie mit der Produkt-basierten Strategie, wird in dieser Arbeit zwischen zwei theoretischen Szenarien unterschieden. In Szenario A wird bei der Produkt-basierten Strategie immer nur ein Produkt verifiziert und das nächste Produkt erst nach Beendigung der Verifikation des vorherigen Produkts analysiert. In diesem Szenario bildet damit der verfügbare Speicher die physische Grenze, wohingegen der zeitliche Aspekt unberücksichtigt bleibt. Dieses Szenario wird in dieser Arbeit als *sequenzielles Produkt-basiertes Model Checking* bezeichnet. In Szenario A ist der Zustandsraum der Familien-basierten Strategie größer als der Zustandsraum der Produkt-basierten Strategie. Szenario B stellt den Gegensatz zu Szenario A dar. In Szenario B werden bei der Produkt-basierten Strategie alle Produkte gleichzeitig verifiziert, wodurch die Größe des Zustandsraums aus der Summe aller Zustandsräume der einzelnen Produkte gebildet wird. Szenario B wird in dieser Arbeit als *paralleles Produkt-basiertes Model Checking* bezeichnet. In diesem Fall ist der Zustandsraum der Familien-basierten Strategie kleiner als der Zustandsraum der parallelen Produkt-basierten Strategie. Der Zustandsraum der Feature-basierte Strategie hingegen ist generell deutlich kleiner als der Zustandsraum der Produkt-basierten Strategie.

3.2. Detaillierte Problemstellung

In den letzten Jahren wurden viele Ansätze für statische Verifikation entwickelt, die diese verschiedenen Verifikationsstrategien verfolgen. Diese Ansätze adaptieren unterschiedliche Analysetechniken wie Model Checking, Type Checking, Static Analysis und Theorem Proving für Software-Produktlinien. Diese möglichen Analysetechniken und Verifikationsstrategien bieten viele mögliche Kombinationen. Zusätzlich unterscheiden sich Software-Produktlinien in Merkmalen wie der Anzahl Produkte, der Anzahl Features und der Codegröße. Aus diesen Gründen wurde die folgende Forschungsfrage ermittelt, die in dieser Arbeit untersucht wird.

> **F1**: Welche Verifikationsstrategie ist signifikant besser oder schlechter für Software-Produktlinien mit gewissen Eigenschaften?

Die Forschungsfrage F1 bezieht sich auf den Aufwand der Analyse. Mit dieser Frage soll beantwortet werden, welche Analysestrategie den Aufwand in Form von benötigter Zeit oder benötigtem Arbeitsspeicher für welche Software-Produktlinien signifikant reduziert oder erhöht. Signifikanz bedeutet dabei, dass die Unterschiede des Aufwands der verglichenen Verifikationsstrategien nicht zufällig entstanden sind, sondern bis zu einer bestimmten Fehlerwahrscheinlichkeit allgemein gültig sind.

Um diese Frage F1 untersuchen zu können, werden empirische Evaluierungen der entwickelten Analysemethoden benötigt. Jedoch enthalten wissenschaftliche Arbeiten zu den entwickelten Analysetechniken nicht immer empirische Evaluierungen zum benötigten Aufwand. Die Abbildung 3.4 stellt dar, wie viele der von Thüm et al. [79] untersuchten Arbeiten empirische Evaluierungen enthalten. Dabei sind zusätzliche Arbeiten, wie technische Reporte, welche eventuelle Evaluierungen zu Ansätzen ohne Evaluierung aus der Klassifizierung von Thüm et al. [79] enthalten nicht mit in der Abbildung 3.4 berücksichtigt.

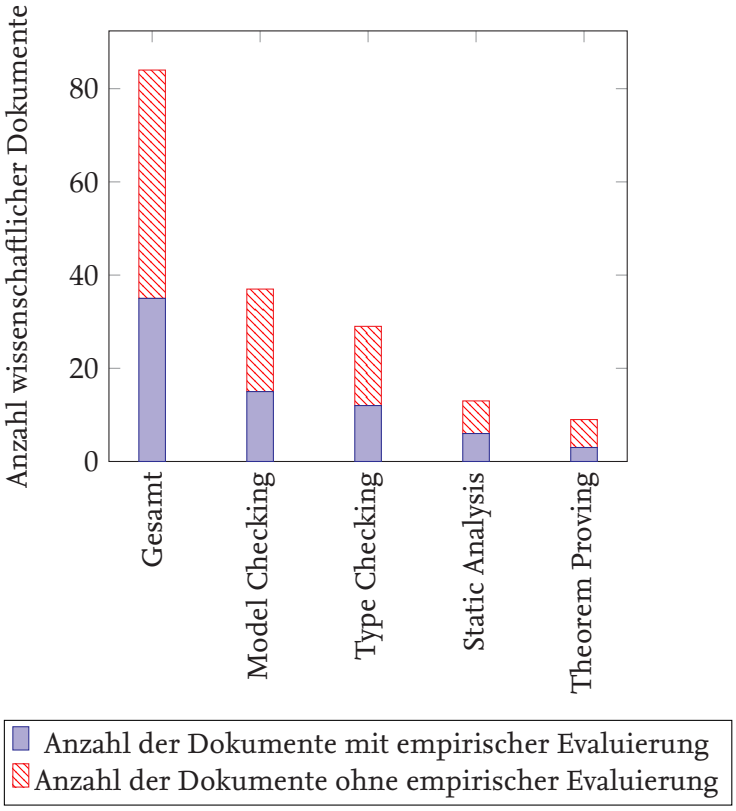


Abbildung 3.4.: Darstellung der wissenschaftlichen Dokumente, die von Thüm et al. [79] berücksichtigt wurden und die Menge dieser Dokumente, die eine empirische Evaluation enthalten.

Außerdem werden in den wissenschaftlichen Arbeiten, in denen empirische Evaluierungen enthalten sind, die entwickelten Ansätze an einer unterschiedlichen Anzahl an Software-Produktlinien evaluiert. Oftmals wird der neue Analyseansatz nur an einer Software-Produktlinie evaluiert, weniger häufig werden dafür zwei Software-Produktlinien genutzt und selten werden mehr als zwei Software-Produktlinien für die Evaluierung verwendet. Somit ergeben sich anhand den Evaluierungen oftmals nur Aussagen für spezifische Software-Produktlinien. Mit einer größeren Anzahl Software-Produktlinien mit Unterschieden in der Anzahl der Features, Produkte und Lines of Code wäre eine eventuelle allgemeinere Aussage pro Evaluierung möglich. Abbildung 3.5 veranschaulicht beispielhaft für die Primärstudien des Model Checkings, wie viele Software-Produktlinien für die empirischen Evaluierungen der Model-Checking-Ansätze verwendet wurden.

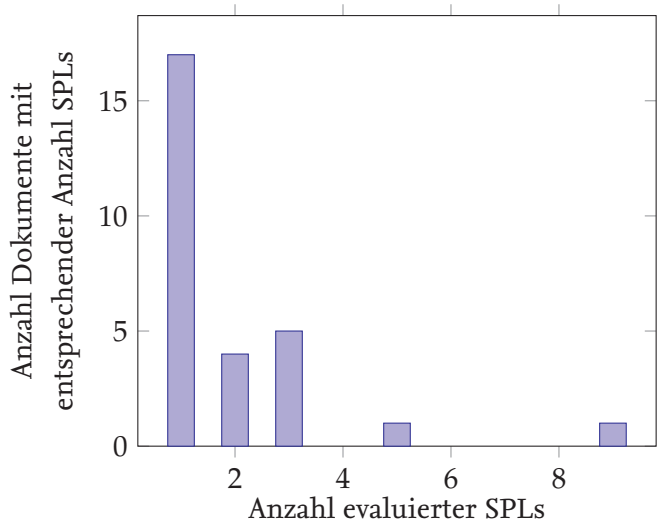


Abbildung 3.5.: Anzahl evaluierter Software-Produktlinien der Model Checking Primärstudien

Ein weiteres Hindernis für die Beantwortung der Frage F1 sind die in den Evaluierungen gegeneinander evaluierten Verifikationsstrategien. Die meisten wissenschaftlichen Dokumente vergleichen die neu entwickelte statische Analyse für Software-Produktlinien mit einer unoptimierten Produkt-basierten Strategie. Beispielhaft dafür ist die wissenschaftliche Arbeit *Detection of Feature Interactions using Feature-Aware Verification* [74]. In der Arbeit von Apel et al. [74] wird die Effizienz der Familien-basierten Analyse eines Metaprodukts, welches alle Features enthält, im Vergleich zum unoptimierten Produkt-basierten Ansatz untersucht. Dies ermöglicht meistens nur einen Vergleich mit der Produkt-basierten Strategie, welchen Mehrwert eine Strategie gegenüber den anderen Strategien besitzt, bleibt dabei unklar.
Weiterhin existieren keine standardisierten Performanz-Metriken, wodurch der Vergleich der einzelnen Evaluierungsergebnisse erschwert wird. Die meisten Studien evaluieren die Verifikationszeit in Sekunden oder Millisekunden, es existieren jedoch beispielsweise auch Evaluierungen in denen der zeitliche Aufwand in der Anzahl der Funktionsaufrufe gemessen wird. Der evaluierte Speicherverbrauch hingegen wird beim Model Checking entweder in der Anzahl der explorierten Zustände oder in Bytes gemessen.
Ein Bezug auf der Performanz auf die Anzahl an unabhängigen Variablen wird zusätzlich dadurch beschränkt, dass bei den Evaluierungen teilweise die Angaben zu den betrachteten unabhängigen Variablen unvollständig sind oder komplett fehlen.

3.3. Untersuchte Dokumente

Die Grundlage der untersuchten Ansätze bilden diejenigen aus der Klassifizierung von Thüm et al. [79]. In der Arbeit von Thüm et al. [79] wurden insgesamt 123 wissenschaftliche Dokumente betrachtet. Diese betrachteten Dokumente enthielten auch verschiedene Versionen des selben Dokuments wie beispielsweise technische Berichte. Nach Entfernen der älteren Versionen wurden insgesamt 84 wissenschaftliche Dokumente in die einzelnen Verifikationsstrategien eingeteilt. Die von Thüm et al. [79] betrachteten Dokumente wurden für diese Arbeit darauf untersucht, welche der Ansätze empirische Evaluierungen der neu entwickelten Analysemethoden enthalten. Die Klassifi-

zierung enthält 35 Ansätze mit dem Fokus auf Model Checking von Software-Produktlinien. Diese 35 Dokumente wurden einer genauen Prüfung darauf unterzogen, ob sie empirische Evaluierungen zu den vorgestellten Ansätzen enthalten. So wurden 16 Dokumente bezüglich Model Checking ermittelt. Diese Dokumente werden im folgenden als *primäre Dokumente* bezeichnet. Des Weiteren ergaben sich daraus 12 primäre Dokumente bezüglich Type Checking, 6 primäre Dokumente bezüglich Static Analysis und drei bezüglich Theorem Proving.
Im darauffolgenden Schritt wurde die Techniken *Backward-Snowballing* und *Forward-Snowballing* in Verbindung mit den primären Dokumenten verwendet [63]. Dazu wurde die Suchmaschine *Google Scholar* verwendet. Mittels Backward-Snowballing wurden die Dokumente ermittelt, die in den primären Dokumenten als Referenzen aufgeführt wurden. Dadurch werden nur ältere Dokumente ermittelt. Forward-Snowballing hingegen findet nur neuere Dokumente, die das jeweilige primäre Dokument zitieren. Außerdem wurden Dokumente untersucht, die von Google Scholar als *ähnliche Dokumente* eingestuft wurden. Dabei wurde die Suche auf die 100 ähnlichsten Ergebnisse begrenzt. Zusätzlich wurde ein systematischer Prozess verwendet, um weitere Ansätze mit empirischen Evaluierungen zu ermitteln. Dazu wurden die folgenden Datenbanken und Suchmaschinen verwendet.

- Google Scholar
- IEEExplore
- SpringerLink
- ACM
- ScienceDirect
- DBLP
- Scopus

Für den Literaturrechercheprozess wurden Suchbegriffe und Suchphrasen festgelegt. Diese Schlüsselwörter wurden in primäre und sekundäre Begriffe unterteilt. Primäre Suchbegriffe sind für alle Teilprozesse der Recherche für die jeweiligen Bereiche Model Checking, Type Checking, Static Analysis diesselben. Die primären Suchbegriffe wurden anhand der ermittelten gebräuchlichen Bezeichnungen für Software-Produktlinien sowie den verwendeten Schlüsselwörtern der Dokumente aus der Arbeit von Thüm et al. [79] ausgewählt. Dahingegen sind die sekundären Schlüsselwörter für jeden dieser Teilbereiche individuell. Die primären Suchbegriffe und Phrasen sind nachfolgend aufgelistet.

- “software product line*“
- “product family“
- “program family“
- “software family“
- “software variants“

3.3.1. Model Checking

Zur Ermittlung von Ansätzen zum Model Checking wurden für diese Analysetechnik spezifische Suchbegriffe und -phrasen festgelegt. Diese spezifischen Schlüsselwörter sind in der folgenden Auflistung dargestellt.

- “model check*“
- LTL
- CTL
- “temporal logic"
- FTS
- MTS
- LTS

Aus den Suchbegriffen wurde folgende Suchanfrage gebildet.

(“software product line“ **OR** “product family“ **OR** “software family“ **OR** “software variants“) **AND** (“model check*“ **OR** LTL **OR** CTL **OR** “temporal logic“ **OR** FTS **OR** MTS **OR** LTS)

Bei den Datenbanken und Suchmaschinen, die keine Verwendung mehrerer boolescher Operatoren zur Verknüpfung der Suchbegriffe unterstützen, wurden die Suchbegriffe iterativ in folgender Art und Weise verwendet.

Primärer Suchbegriff **AND** *Sekundärer Suchbegriff*

In diesem Prozess der Literaturrecherche wurden nur Dokumente aus den Jahren 2014 und 2015 berücksichtigt. Die Recherche ergab insgesamt 2985 potenzielle Dokumente. Aus den Suchergebnissen wurden die Dokumente entfernt, die die Begriffe *test* oder *testing* im Titel oder Abstract enthalten. Diese Dokumente wurden ausgeschlossen, da der Fokus dieses Arbeit auf formalen Verifikationsmethoden liegt und dynamische Verifikationsverfahren wie Testen nicht berücksichtigt. Dadurch wurde die Anzahl der potenziellen Studien auf 1089 reduziert. Anschließend wurden aus diesen 1089 potenziellen Primärstudien diejenigen Dokumente ausgewählt, deren Titel sowohl eine Bezeichnung für Software-Produktlinien als auch zum Model Checking verwandte Begriffe enthielt. Diese Auswahl ergab 401 potenzielle Dokumente, bei denen im nächsten Schritt die Duplikate entfernt wurden. Die Entfernung von Duplikaten resultierte in 69 potenziellen Dokumenten. Auf eines der potenziellen Dokumente konnte nicht zugegriffen werden, da keine Berechtigung zum Zugriff auf die chinesische Datenbank vorlag, in der es publiziert wurde. Die restlichen 68 Dokumente wurden daraufhin auf enthaltene empirische Daten untersucht und dieser Vorgang reduzierte die Anzahl an potenziellen Dokumenten auf zwölf. Die 16 Ansätze der Klassifizierung und die zwölf potenziellen Dokumente wurden im letzten Schritt detailliert geprüft und die relevanten Daten für die Untersuchungen aus Kapitel 3.4 extrahiert. In dieser Arbeit wurden somit insgesamt 28 wissenschaftliche Dokumente bezüglich Model Checking berücksichtigt, die verschiedene Ansätze basierend auf den vorgestellten Analysestrategien vorstellen und empirische Evaluierungen

dazu enthalten. Die untersuchten Ansätze unterscheiden sich in vielen Merkmalen, nicht nur in den verwendeten Verifikationsstrategien. Einige der Ansätze verwenden LTL-Logik zur Beschreibung der zu verifizierenden Eigenschaften und einige Ansätze verwenden CTL-Logik. Einige Ansätze wiederum erweitern diese Logik-Klassen um Variabilitätsoperatoren zur Formulierung von Eigenschaften für Teilmengen der Produkte der Software-Produktlinien. Außerdem wird das zu verifizierende System in den Ansätzen mit unterschiedlichen Techniken modelliert. Beispielsweise wurden in der Arbeit von Classen et al. [8] Transitionssysteme um die Repräsentation von Variabilität erweitert. Diese erweiterten Transitionssysteme werden *Featured Transition Systems* (*FTS*) genannt. Weitere Beispiele für Modellierungstechniken sind *Modale Sequenzdiagramme* (*MSD*) [40], *Actor Families* [30] oder die auf Software-Produktlinien adaptierte Algebra *PL-CCS* [29].
Unter den 28 Dokumenten sind Ansätze enthalten, welche symbolische Methoden und weitere Abstraktionstechniken zur Reduzierung des Systemmodells verwenden oder Ansätze, die auf Wahrscheinlichkeits-basiertem Model Checking basieren. Einige der Ansätze präsentieren neue Model-Checking-Algorithmen und Model Checker, welche die Variabilität von Software-Produktlinien berücksichtigen. Eine Zusammenfassung der einzelnen Ansätze ist im Anhang A im Kapitel A.1.1 verfügbar.
Die Tabelle 3.7 bietet eine Übersicht über die verwendeten wissenschaftlichen Arbeiten für die vorliegende Arbeit. In der Tabelle 3.7 sind alle sich unterhalb der breiten horizontalen Linie befindenden wissenschaftlichen Arbeiten neu ermittelte Primärstudien.

Kürzel	Titel der wissenschaftlichen Arbeit
MC-A 1	Detection of Feature Interactions using Feature-Aware Verification [74]
MC-A 2	Features meet Scenarios: Modeling and Consistency-Checking Scenario-Based Product Line Specifications [40]
MC-A 3	Simulation-Based Abstractions for Software Product-Line Model Checking [55]
MC-A 4	Compositional Algorithmic Verification of Software Product Lines [38]
MC-A 5	Efficient Verification of Evolving Software Product Lines [29]
MC-A 6	Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features [58]
MC-A 7	Formal Semantics, Modular Specification, and Symbolic Verification of Product-Line Behaviour [7]
MC-A 8	Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and their Application to LTL Model Checking [8]
MC-A 9	Configuration Lifting: Verification meets Software Configuration [32]
MC-A 10	Strategies for Product-Line Verification: Case Studies and Experiments [71]
MC-A 11	Behavioural Modelling and Verification of Real-Time Software Product Lines [57]
MC-A 12	Model Checking lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines [10]
MC-A 13	Modeling and Verification of Reconfigurable Actor Families [30]
MC-A 14	Reducing the Verification Cost of Evolving Product Families using Static Analysis Techniques [31]
MC-A 15	Compositional Model Checking of Software Product Lines using Variation Point Obligations [38]
MC-A 16	Towards Formal Safety Analysis in Feature-Oriented Product Line Development [64]
MC-A 17	Coherent Branching Feature Bisimulation [75]
MC-A 18	Probabilistic Model Checking for Energy Analyis in Software Product Lines [26]
MC-A 19	Counterexample Guided Abstraction Refinement of Product-Line Behavioural Models [56]
MC-A 20	Modeling and Verification for Probabilistic Properties in Software Product Lines [28]
MC-A 21	Probabilistic Model Checking for Feature-Oriented Systems [25]
MC-A 22	Requirements Verification of Variability-Intensive Systems [6]
MC-A 23	Family-Based Model Checking without a Family-Based Model Checker [2]
MC-A 24	Incremental Model Checking of Delta-Oriented Software Product Lines [50]
MC-A 25	Model Checking of Domain Artifacts in Product Line Engineering [45]
MC-A 26	Model-Based Verification of Quantitative Non-Functional Properties for Software Product Lines [15]
MC-A 27	Symbolic Model Checking of Product-Line Requirements using SAT-Based Methods [68]
MC-A 28	Potential Synergies of Theorem Proving and Model Checking in Software Product Lines [78]

Tabelle 3.7.: Verwendete wissenschaftliche Arbeiten zu Model Checking für die Evaluierung der Verifikationsstrategien

3.3.2. Type Checking

Auch für die Ermittlung von Ansätzen für Type Checking von Software-Produktlinien wurden vor dem Suchprozess Suchbegriffe und Suchphrasen festgelegt. Die spezifischen Suchbegriffe für den Bereich Type Checking sind in der folgenden Auflistung aufgeführt. Diese Begriffe wurden während der Studie der primären Dokumente bezüglich Type Checking aus der Studie von Thüm et al. [79] ermittelt und repräsentieren übliche Bezeichnungen für das Type Checking. In der nachfolgenden Auflistung sind die Phrasen durch Anführungszeichen gekennzeichnet.

- “type check*“
- “type system“
- “safe composition“
- “dangling reference*"

Aus den Suchbegriffen wurde folgende Suchanfrage gebildet.

(“software product line“ **OR** “product family“ **OR** “software family“ **OR** “software variants“) **AND** (“type check*“ **OR** “type system“ **OR** “safe composition“ **OR** “dangling reference*“)

Der Literatur-Rechercheprozess erfolgte analog zu dem in Abschnitt 3.3.1 beschriebenen Rechercheprozess. Dadurch wurden sechs zusätzliche Primärstudien ermittelt, wodurch insgesamt 18 wissenschaftliche Dokumente bezüglich Type Checking berücksichtigt wurden. In diesen 18 Primärstudien werden unterschiedliche Ansätze zum Type Checking von Software-Produktlinien basierend auf den vorgestellten Analysestrategien vorgestellt und mittels Experimenten und Fallstudien empirisch evaluiert. Eine Zusammenfassung der einzelnen Ansätze ist im Anhang A im Kapitel A.1.2 verfügbar. Die Tabelle 3.9 bietet eine Übersicht über die verwendeten wissenschaftlichen Arbeiten für die vorliegende Arbeit. Durch den Rechercheprozess ermittelte neue Primärstudien sind in der Tabelle 3.9 unterhalb der breiten horizontalen Linie aufgelistet.

Kürzel	Titel der wissenschaftlichen Arbeit
TC-A 1	A Comparison of Product-based, Feature-based, and Family-based Type Checking [65]
TC-A 2	Safe Composition of Configuration Knowledge-based Software Product Lines [49]
TC-A 3	Safe Composition of Product Lines [62]
TC-A 4	Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints [47]
TC-A 5	Extending Type Inference to Variational Programs [66]
TC-A 6	Safe Composition of Non-Monotonic Features [52]
TC-A 7	TypeChef: Toward Type Checking #ifdef Variability in C [11]
TC-A 8	Scalable Analysis of Variable Software [43]
TC-A 9	Supporting Consistency Checking between Features and Software Product Line Use Scenarios [54]
TC-A 10	A Variability-Aware Module System [19]
TC-A 11	Type Checking Annotation-Based Product Lines [20]
TC-A 12	Configuration Lifting Software: Verification meets Software Configuration [32]
TC-A 13	Experiments on Type Checking of Software Product Lines [24]
TC-A 14	Type-based parametric analysis of program families [67]
TC-A 15	Analysis and Transformation of Configurable Systems [41]
TC-A 16	Typprüfung von Produktlinien in FUJI [61]
TC-A 17	Large-Scale Variability Aware Type Checking and Dataflow Analysis [42]
TC-A 18	Typsicherheit in Feature-orientierten Software-Produktlinien in FeatureIDE [69]

Tabelle 3.9.: Verwendete wissenschaftliche Arbeiten zu Type Checking für die Evaluierung der Verifikationsstrategien

3.3.3. Static Analysis

Für die Literaturrecherche von Static-Analysis-Ansätzen für Software-Produktlinien erfolgte ebenfalls eine Festlegung der Suchbegriffe und Suchphrasen. Diese Suchbegriffe für die statische Analyse sind nachfolgend aufgezählt. Diese Suchkomponenten wurden durch Untersuchung der Ansätze bezüglich Static Analysis von Software-Produktlinien aus der Klassifizierung von Thüm et al. [79] gebildet. Dabei wurden ausschließlich die drei folgenden Suchbegriffe berücksichtigt, da in dieser Arbeit mit dem Begriff *Static Analysis* hauptsächlich Datenflussanalysen bezeichnet werden. Suchphrasen sind in der folgenden Aufzählung durch Anführungszeichen markiert.

- “static analysis“
- “dataflow“
- “control flow“

Aus den Suchbegriffen wurde folgende Suchanfrage gebildet.

(“software product line“ **OR** “product family“ **OR** “software family“ **OR** “software variants“) **AND** (“static analysis“ **OR** “dataflow“ **OR** “control flow“)

Der Literatur-Rechercheprozess und die Ausschlusskriterien für potenzielle Studien sind analog zu dem Prozess und den Kriterien aus Kapitel 3.3.1. Die systematische Literaturrecherche resultierte

in fünf weiteren Primärstudien zum Vergleich der Verifikationsstrategien bezüglich Static Analysis von Software-Produktlinien. Zusammen mit den sechs Primärstudien aus der Arbeit von Thüm et al. [79] wurden elf Primärstudien verwendet, in denen unterschiedliche Ansätze empirisch evaluiert wurden, verwendet. Eine Kurzfassung der einzelnen Ansätze ist im Anhang A im Kapitel A.1.3 aufgeführt. Die Tabelle 3.11 listet die verwendeten wissenschaftlichen Arbeiten zur Evaluierung von Verifikationsstrategien in Kombination mit Static Analysis für die vorliegende Arbeit auf. Die neu ermittelten Primärstudien sind in der Tabelle 3.11 unterhalb der breiten horizontalen Trennlinie aufgeführt.

Kürzel	Titel der wissenschaftlichen Arbeit
SA-A 1	Intraprocedural Dataflow Analysis for Software Product Lines [23]
SA-A 2	SPLLIFT: Statically Analyzing Software Product Lines in Minutes instead of Years [27]
SA-A 3	Integration Testing of SPLs Using Compositional Symbolic Execution [37]
SA-A 4	Reducing Combinatorics in Testing Product Lines [16]
SA-A 5	Reducing Configurations to Monitor in a Software Product Line [17]
SA-A 6	Reducing the Verification Cost of Evolving Product Families using Static Analysis Techniques [31]
SA-A 7	Systematic Derivation of Correct Variability-Aware Program Analyses [34]
SA-A 8	Variability Abstractions: Trading Precision for Speed in Family-Based Analyses [3]
SA-A 9	Analysis and Transformation of Configurable Systems [41]
SA-A 10	Large-Scale Variability Aware Type Checking and Dataflow Analysis [42]
SA-A 11	Scalable Analysis of Variable Software [43]

Tabelle 3.11.: Verwendete wissenschaftliche Arbeiten zu Static Analysis für die Evaluierung der Verifikationsstrategien

3.3.4. Theorem Proving

Die Literaturrecherche zur Suche weiterer Primärstudien zur Evaluierung von Verifikationsstrategien für Software-Produktlinien in Verbindung mit Theorem Proving erfolgte anhand vorher definierter Suchbegriffe und Suchphrasen. Dabei wurden diese relevanten Begriffe aus den Theorem-Proving-Ansätzen aus der Klassifikation von Thüm et al. [79] ermittelt. Diese Suchbegriffe enthalten gebräuchliche Bezeichnungen des Theorem Provings. Für den Suchprozess bezüglich Theorem Proving wurden ausschließlich Suchphrasen verwendet, die in der folgenden Auflistung durch Anführungszeichen gekennzeichnet sind.

- “theorem proving“
- “deductive verification“

Aus den Suchbegriffen wurde folgende Suchanfrage gebildet.

(“software product line“ **OR** “product family“ **OR** “software family“ **OR** “software variants“) **AND** (“theorem proving“ **OR** “deductive verification“)

Durch den Literatur-Rechercheprozess wurden drei weitere Primärstudien ermittelt. Dabei erfolgte die Auswahl und der Ausschluss von potenziellen Studien analog zur Beschreibung des Re-

chercheprozess aus Kapitel 3.3.1. Insgesamt wurden somit sechs wissenschaftliche Dokumente bezüglich Theorem Proving verwendet, in denen unterschiedliche Ansätze zum Theorem Proving von Software-Produktlinien basierend auf den vorgestellten Analysestrategien empirisch evaluiert werden. Eine Zusammenfassung der einzelnen Ansätze ist im Anhang A im Kapitel A.1.4 verfügbar. Die Tabelle 3.13 bietet eine Übersicht über die verwendeten wissenschaftlichen Arbeiten für die vorliegende Arbeit. Primärstudien, die durch die Literaturrecherche neu ermittelt wurden, befinden sich in der Tabelle 3.13 unterhalb der breiten horizontalen Trennlinie.

Kürzel	Titel der wissenschaftlichen Arbeit
TP-A 1	Family-Based Deductive Verification of Software Product Lines [77]
TP-A 2	Proof Composition for Deductive Verification of Software Product Lines [76]
TP-A 3	Product Lines of Theorems [13]
TP-A 4	Effiziente Kodierung von Variabilität in Spezifikationen [53]
TP-A 5	JML-Based Verification for Feature-Oriented Programming [36]
TP-A 6	Potential Synergies of Theorem Proving and Model Checking in Software Product Lines [78]

Tabelle 3.13.: Verwendete wissenschaftliche Arbeiten zu Theorem Proving für die Evaluierung der Verifikationsstrategien

3.4. Hypothesen

Im Kapitel 3.2 wurde die Forschungsfrage F1 formuliert, welche Strategie die effizienteste für Software-Produktlinien mit spezifischen Eigenschaften ist. Für diese Forschungsfrage werden im Folgenden Hypothesen erstellt, die in Kapitel 4 untersucht werden. Dabei wird versucht, die Richtigkeit der Hypothesen unter Verwendung von empirischen Evaluierungen der wissenschaftlichen Dokumente durch Diagrammen und Signifikanztests zu ermitteln.

Dabei werden sowohl abhängige als auch unabhängige Variablen betrachtet. Die unabhängigen Variablen spezifizieren die spezifischen Eigenschaften einer Software-Produktlinie. Unabhängige Variablen können in der Evaluierung geändert werden und bezeichnen somit die Variablen, in denen sich die einzelnen evaluierten Software-Produktlinien voneinander unterscheiden. Zu diesen unabhängigen Variablen gehören die Anzahl der Features, die Anzahl der Produkte sowie die Codegröße der evaluierten Software-Produktlinie. Diese unabhängigen Variablen wurden ausgewählt, da sie die drei intuitivsten Merkmale einer Software-Produktlinie darstellen.

Abhängige Variablen verändern sich in Abhängigkeit einer oder mehrerer unabhängiger Variablen. Die in dieser Studie betrachteten abhängigen Variablen sind die Verifikationszeit und der Speicherverbrauch der einzelnen Analysestrategien. Eine Beschränkung auf diese beiden abhängigen Variablen erfolgte anhand der betrachteten Verifikationsaufwände der einzelnen Evaluierungen. Die Tabelle 3.15 fasst die in dieser Arbeit betrachteten Variablen zusammen.

Name der Variable	Art der Variable	Beschreibung
Anzahl Features	Unabhängige Variable	Variable der Software-Produktlinie - repräsentiert die Menge der Features
Anzahl Produkte	Unabhängige Variable	Variable der Software-Produktlinie - repräsentiert die Menge möglicher Produkte
Lines of Code (*LoC*)	Unabhängige Variable	Variable der Software-Produktlinie - repräsentiert die Anzahl der Quellcode-Zeilen der gesamten Produktlinie
Verifikationszeit	Abhängige Variable	Variable der Verifikationsmethode - beschreibt die Dauer der statischen Analyse
Speicherverbrauch	Abhängige Variable	Variable der Verifikationsmethode - beschreibt den benötigten Speicher zur Durchführung der statischen Analyse

Tabelle 3.15.: Abhängige und unabhängige Variablen, die in dieser Arbeit betrachtet werden

3.4.1. Hypothesen der untersuchten Paper

In einigen der untersuchten wissenschaftlichen Dokumente stellen die Autoren Hypothesen bezüglich der Effizienz der verschiedenen Analysestrategien auf. Die Grundlage der in dieser Arbeit formulierten Hypothesen bilden Behauptungen zum Aufwand der Strategien von Thüm et al. [79]. In der Arbeit von Thüm et al. sind folgende Behauptungen enthalten beziehungsweise lassen sich folgende Aussagen ableiten.

Behauptung 1 Der Aufwand der Familien-basierten Strategie ist abhängig von der Codegröße der Implementierungsartefakte sowie der Wiederverwendung von Ergebnissen während der Analyse [23].

Behauptung 2 Die Familien-basierte Strategie ist durch physische Grenzen wie beispielsweise dem verfügbaren Arbeitsspeicher beschränkt und stößt oftmals an diese Grenzen [71].

Behauptung 3 Bei weiterentwickelten Software-Produktlinien ist der benötigte Aufwand der Familien-basierten und Produkt-basierten Strategie sehr groß. In diesem Fall kann die Feature-basierte Strategie eine Alternative mit deutlich geringerem Aufwand darstellen.

Behauptung 4 Der Aufwand der Familien-basierte Strategie ist nicht proportional zur Anzahl der Produkte.

Aussage 1 Die Familien-basierte Strategie besitzt einen geringeren Verifikationsaufwand als die Produkt-basierte Strategie aufgrund der Vermeidung von redundanten Berechnungen.

Aussage 2 Für eine gegebene Software-Produktlinie ist der Aufwand der Produkt-basierten Strategie proportional zur Anzahl der Produkte der Produktlinie.

In den Primärstudien zu den betrachteten Ansätzen bezüglich der formalen Verifikationstechniken Model Checking, Type Checking, Static Analysis und Theorem Proving wurden ebenfalls Hypothe-

sen bezüglich der Performanz der verschiedenen Verifikationsstrategien formuliert. Diese Hypothesen sind in der folgenden Aufzählung erläutert.

PH1 Apel et al. [71] tätigen die Annahme, dass ein Sample-basierter Produkt-basierter Ansatz schneller als der Familien-basierte Ansatz sein kann, dabei jedoch eventuell nicht alle verletzten Eigenschaften der Software-Produktlinie aufdeckt.

PH2 Kolesnikov et al. [65] stellen die Hypothese auf, dass der Familien-basierte Ansatz schneller ist als die Produkt-basierte Strategie. Sie stellen dazu eine zweite Hypothese als Ausnahme für die erste Hypothese auf, dass bei Software-Produktlinien mit wenigen Produkten die Produkt-basierte Strategie schneller ist als die Familien-basierte Strategie.

PH3 Eine weitere Hypothese von Kolesnikov et. al [65] besagt, dass für Software-Produktlinien mit wenigen Features die Feature-basierte Strategie schneller ist als die Familien-basierte Strategie.

PH4 Teixeira et al. [49] behaupten, dass der Vorteil der Familien-basierten Strategie je größer ist desto größer die Software-Produktlinie ist. Es wird in dieser Primärstudie jedoch nicht erläutert, ob die Größe einer Software-Produktlinie als Anzahl der Features, der Produkte oder der Codezeilen spezifiziert ist.

PH5 In der Primärstudie von Thaker et al. [62] wird lediglich behauptet, dass die Produkt-basierte Strategie unpraktikabel ist.

PH6 Chen et al. [66] behaupten zum einen, dass die Produkt-basierte Strategie aus Effizienzgründen oftmals nicht durchführbar ist. Zum anderen behaupten sie, dass die Verifikationsergebnisse der Produkt-basierten Strategie nicht gut auf die Variabilität der Software-Produktlinien abgebildet werden können.

PH7 In der Studie von Liebig et al. [43] wird die Hypothese formuliert, dass die optimierte Produkt-basierte Strategie in Form der *Single Conf Sampling-Strategie* schneller ist als die Familien-basierte Strategie. Die zweite Hypothese von Liebig et al. [43] besagt, dass der Familien-basierte Ansatz schneller als die optimierte Produkt-basierte Strategie in Form der *Pair-Wise Sampling-Strategie* ist.

3.4.2. Ermittelte Hypothesen

Die Hypothesen bezüglich der Performanz der verschiedenen Verifikationsstrategien werden in diesem Abschnitt dargestellt. Die Hypothesen beziehen sich hauptsächlich auf den Vergleich zwischen der Familien-basierten Strategie und der Produkt-basierten Strategie, da dies der hauptsächlich durchgeführte Evaluierungsvergleich der einzelnen Primärstudien ist. Vereinzelt sind Ansätze zu weiteren Strategien vorhanden, deren geringe Frequenz jedoch keine Überprüfung von Hypothesen ermöglicht. Als *globale Hypothesen* werden die Hypothesen bezeichnet, die für alle der in dieser Arbeit berücksichtigten statischen Analysen angenommen werden. Diese Hypothesen werden zuerst für die jeweiligen Analysetechniken überprüft und anschließend die Allgemeingültigkeit für alle betrachteten formalen Analysemethoden interpretiert.
Die Hypothesen wurden größtenteils von Behauptungen und Hypothesen aus den untersuchten

wissenschaftlichen Arbeiten abgeleitet. Die Behauptungen, aus denen die jeweilige Hypothese abgeleitet wurde, ist hinter der jeweiligen globalen Hypothese markiert. Wenn anschließend an eine Hypothese keine ursprüngliche Behauptung annotiert ist, wurde diese Hypothese aus den Erkenntnissen der Untersuchungen der Primärstudien hergeleitet. Durch die Untersuchung dieser Hypothesen in Kapitel 3.4 wird versucht, die Auswahl der besten Verifikationsstrategie für eine gegebene Software-Produktlinie zu erleichtern. Des Weiteren soll durch die Ergebnisse der Hypothesenuntersuchungen gezeigt werden, welche vermuteten Vor- und Nachteile die einzelnen Strategien in Bezug auf die abhängigen Variablen besitzen.

Globale Hypothesen Die globalen Hypothesen beziehen sich auf alle der in dieser Arbeit betrachteten statischen Analysemethoden.

- GH1: Für viele Produkte ist die Familien-basierte Strategie schneller als die Produkt-basierte Strategie (PH2).
- GH2: Für wenige Produkte ist der Produkt-basierte Ansatz schneller als der Familien-basierte Ansatz (PH2).
- GH3: Für viele Features ist die Familien-basiertes Strategie schneller als die Produkt-basierte Strategie (PH2).
- GH4: Für wenige Features ist die Produkt-basierte Strategie schneller als die Familien-basierte Strategie (PH2).
- GH5: Für Software-Produktlinien mit vielen *LoC* ist der Familien-basierte Ansatz schneller als der Produkt-basierte Ansatz (Behauptung 1, PH4).
- GH6: Der Speicherverbrauch der Familien-basierten Strategie ist geringer als der Speicherverbrauch der parallelen Produkt-basierten Strategie.
- GH7: Der Speicherverbrauch der sequenziellen Produkt-basierten Strategie ist geringer als der Speicherverbrauch der Familien-basierten Strategie.
- GH8: Je höher die Gemeinsamkeiten der Produkte einer SPL, desto höher ist das Einsparungspotential durch den Familien-basierten Ansatz (Aussage 1).

3.5. Zusammenfassung

In diesem Kapitel wurden die verschiedenen Verifikationsstrategien für Software-Produktlinien vorgestellt. Anschließend wurde die Problemstellung der vorliegenden Arbeit detailliert erläutert. Beispielsweise wurde gezeigt, dass der Anteil an wissenschaftlichen Dokumenten bezüglich dieser Verifikationsstrategien inklusive empirischer Evaluierungen sehr gering ist. Anhand der Ergebnisse der systematischen Literaturrecherche wurden weitere neuere solcher wissenschaftliche Dokumente gefunden, die zusammen mit den Ansätzen aus der Studie von Thüm et al. [79] die Datengrundlage für die Untersuchungen bilden. Dazu wurden in diesem Kapitel ebenfalls Hypothesen formuliert, die mittels diesen Primärstudien und deren empirischen Evaluierungen in Kapitel 4 untersucht werden.

4 Vergleich der Verifikations -strategien

In diesem Kapitel werden die verschiedenen Hypothesen überprüft, die aus Kapitel 3 resultieren. Dazu werden die Evaluierungsergebnisse der recherchierten Primärstudien verwendet. In den folgenden Unterkapiteln werden die einzelnen Hypothesen bezüglich der betrachteten formalen Verifikationstechniken untersucht. Dabei werden die Untersuchungsergebnisse mittels Diagrammen und Tabellen visualisiert und interpretiert. Am Ende der Hypothesenuntersuchungen werden die Gefährdungen der Validität der Untersuchungen der Primärstudien sowie der in dieser Arbeit durchgeführten Hypothesentests erläutert.
Die Gefahren der Validität der einzelnen Primärstudien sind teilweise in *interne Validität, externe Validität* sowie *Konstruktvalidität* unterteilt. *Interne Validität* stellt die Validität des kausalen Zusammenhangs der Untersuchung dar, wohingegen mittels der *externen Validität* eventuelle Umstände der Untersuchungen dokumentiert werden, die die Allgemeingültigkeit der Erkenntnisse für eine bestimmte Domäne gefährden können [28]. In der Betrachtung der *Konstruktvalidität* wird die Korrektheit der operationalen Messung diskutiert [28].
In Abschnitt 4.2 werden die Hypothesen bezüglich Model Checking untersucht. In den Abschnitten 4.3, 4.4 und 4.5 werden respektive die Hypothesen bezüglich Type Checking, Static Analysis und Theorem Proving getestet. In dieser Arbeit werden erstmals verschiedene Primärstudien bezüglich der Verifikationszeit und des Speicherverbrauchs in einer einzelnen Studie untersucht. Dabei werden zum ersten Mal Aussagen über die Performanz der betrachteten Verifikationsstrategien über verschiedene Ansätze hinweg getätigt. Des Weiteren bieten die Interpretationen der Untersuchungsergebnisse der jeweiligen Hypothesen Bestätigungen bzw. Widerlegungen der jeweiligen Hypothesen. Somit bieten die Ergebnisse der Hypothesentests Aussagen bezüglich eines eventuellen signifikanten Vorteils einer Verifikationsstrategie gegenüber einer anderen Strategie. Am Ende dieses Kapitels werden die Untersuchungsergebnisse der Hypothesen der einzelnen formalen Verifikationstechniken für eine generelle Aussage für Software-Produktlinien in Abschnitt 4.7 zusammengefasst.

4.1. Untersuchungsvorgang & Methoden zur Hypothesenuntersuchung

In diesem Abschnitt wird erläutert, mit welchen Methoden und Techniken die formulierten Hypothesen untersucht werden.
Zu Beginn der Untersuchungen zum Verhältnis der Verifikationszeit zu den unabhängigen Variablen wurden alle verfügbaren Datensätze der Primärstudien in gemeinsame Streudiagramme aggregiert. Diese Streudiagramme werden zum einen in Abhängigkeit von der Anzahl der Produkte,

in Abhängigkeit von der Anzahl der Features und zum anderen in Abhängigkeit der Größe des Quellcodes erstellt. Diese Diagramme werden als Meta-Diagramme bezeichnet. Durch die Darstellung der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit dieser unabhängigen Variablen in einem gemeinsamen Streudiagramm wird eine allgemeine Tendenz dieser Strategien visualisiert.
Nachdem diese Diagramme eine intuitive Darstellung bezüglich der Performanz der Produkt-basierten Strategie und der Familien-basierten Strategie erbringen, werden die einzelnen Primärstudien auf signifikante Verbesserungen durch Verwendung der Familien-basierten Strategie untersucht. Außerdem werden die weiteren Verifikationsstrategien der Primärstudien ebenfalls auf signifikante Unterschiede untersucht. Dazu wird im ersten Schritt ein geeigneter statistischer Test für die vorliegenden Daten ermittelt. Der passende statistische Test für die vorliegenden Datensätze ist der *Wilcoxon-Vorzeichen-Rang-Test*.

4.1.1. Wilcoxon-Vorzeichen-Rang-Test

Der Wilcoxon-Vorzeichen-Rang-Test ist ein Test für sowohl unabhängige als auch abhängige Stichproben. Bei der Untersuchung der einzelnen Primärstudien auf Signifikanz liegen abhängige Stichproben vor. Abhängige Stichproben bedeuten in dem Fall dieser Untersuchung, dass die Produkt-basierte Strategie und die Familien-basierte Strategie für dieselben Software-Produktlinien beziehungsweise Variationen der Software-Produktlinien evaluiert wurden. Die Stichprobengröße N (Anzahl der Datenpaare für Produkt-basierte und Familien-basierte Strategie) muss für den Wilcoxon-Vorzeichen-Rang-Test mindestens gleich fünf sein.
Der Wilcoxon-Vorzeichen-Rang-Test ist ein parameterfreier Signifikanztest. Parameterfrei bedeutet, dass er keine Voraussetzungen wie beispielsweise Normalverteilung erfordert, um verwendet werden zu können (siehe [22]). Dennoch ist der Test auch relativ robust, falls eine Normalverteilung vorliegen sollte ([22]). Da es sich bei den meisten unserer Tests um abhängige Stichproben handelt und eine Überprüfung der Daten mit einem Shapiro-Test keine Normalverteilung zeigt, ist der Wilcoxon-Vorzeichen-Rang-Test der passende Hypothesentest für diese Untersuchung. Der Test besteht allgemein aus den folgenden Teilschritten.

1. Auflistung der Messergebnisse der Kontrollgruppe (meistens *PROD*) und der Experimentalgruppe (meistens *FAM*) in tabellarischer Form.

2. Bestimmung der Differenz zwischen dem Messergebnis der Kontrollgruppe und der Experimentalgruppe.

3. Zuteilung von Rangplätzen anhand der Beträge der ermittelten Differenzen. Dabei bekommt der kleinste Differenzbetrag den kleinsten Rangplatz, der nächst größere Differenzbetrag den nächst höheren Rangplatz. Diese Zuteilung wird in diesem Schema fortgesetzt, bis alle Messergebnisse einen Rangplatz erhalten. Existieren zwei oder mehr Beträge von Differenzen mit demselben Wert, wird für alle diese Beträge der Quotient aus der Summe der theoretischen Rangplätze und der Anzahl gleich großer Beträge gebildet. Dieser Quotient wird anschließend allen jeweils gleich großen Differenzbeträgen als Rang zugeordnet.

4. Berechnung der positiven und negativen Rangsumme, indem jeweils die Beträge der positiven beziehungsweise negativen Differenzen addiert werden. Beispielsweise sind positive Differenzen beim Vergleich der Produkt-basierten und Familien-basierten Strategie die Werte, in

denen die Familien-basierte Strategie schneller ist als die Produkt-basierte Strategie. Analog sind in diesem Beispiel negative Differenzen die Datenpaare, bei denen die Produkt-basierte Strategie schneller ist. Die Beträge der jeweils positiven sowie negativen Differenzen werden jeweils addiert und somit die positive und negative Rangsumme gebildet.

5. Bestimmung des Prüfwerts als Minimum der positiven und negativen Rangsumme.

Bei dem Wilcoxon-Vorzeichen-Rang-Test wird der Wert w berechnet. Dieser entspricht dem Prüfwert dieses Signifikanztests. Dieser Prüfwert wird mit dem kritischen Wert $w_{\alpha,n}$ verglichen, welcher aus entsprechenden Tabellen anhand des Signifikanzniveaus und der Stichprobengröße abgelesen werden kann(siehe [44] S. 402). Ist der Prüfwert kleiner oder gleich dem kritischen Wert, wird die Nullhypothese abgelehnt und die Alternativhypothese angenommen.
Bei der Durchführung des Wilcoxon-Vorzeichen-Rang-Tests wird bei einigen statistischen Software-Werkzeugen zusätzlich der p-Wert berechnet. Dabei können sowohl genaue als auch gerundete p-Werte als Ergebnis resultieren. Gerundete p-Werte werden in dieser Arbeit mit einer vorangestellten Tilde markiert. Im Falle von zwei oder mehr gleichen Rangplätzen berechnet der Wilcoxon-Vorzeichen-Rang-Test standardmäßig gerundete Werte. Es ist jedoch auch eine Berechnung von exakten p-Werten möglich. Der p-Wert stellt die Wahrscheinlichkeit dar, ein solches Messergebnis erneut zu erhalten, wenn die Nullhypothese wahr ist. Ist der p-Wert kleiner als das Signifikanzniveau α, deutet diese Tatsache ebenfalls auf eine Ablehnung der Nullhypothese hin. Für eine ausführlichere Erläuterung des p-Werts wird auf das Buch von Hedderich und Sachs ([44]) verwiesen.
Eine ausführlichere Erklärung des Wilcoxon-Vorzeichen-Rang-Tests ist in den dedizierten Büchern wie „*Angewandte Statistik - Methodensammlung mit R*“([44]) und „*Experimentation in Software Engineering*“([22]) zu finden.

4.1.2. Vote Counting

Der Wilcoxon-Vorzeichen-Rang-Test wird in dieser Arbeit unter anderem dazu genutzt, um die Primärstudien darauf zu untersuchen, ob die Familien-basierte Strategie in den einzelnen Primärstudien signifikant schneller ist als die Produkt-basierte Strategie. Die Ergebnisse dieser Untersuchung werden mittels *Vote Counting*, einer Art Auszählungsmethode, kombiniert. Eine simple Form des Vote Countings ist die Gegenüberstellung von Primärstudien mit signifikanten und nicht signifikanten Untersuchungsergebnissen [51]. Vote Counting berücksichtigt den Stichprobenumfang der einzelnen Primärstudien nicht. Im Allgemeinen liefern größere Stichproben jedoch aussagekräftigere Ergebnisse, da durch entsprechend große Stichproben die Auswirkung von Zufallseinflüssen verringert wird. Aufgrund der fehlenden Berücksichtigung des Stichprobenumfangs werden beim Vote Counting alle Ergebnisse gleich stark gewichtet. Dies spricht im Allgemeinen gegen die Verwendung von Methoden zum Vote Counting. In dieser Untersuchung jedoch wird Vote Counting dazu verwendet, um die signifikanten und nicht signifikanten Evaluierungsergebnisse der Primärstudien auszuzählen und gegenüberzustellen. Da der Stichprobenumfang nicht relevant ist für die Auszählung, welche der Primärstudien eine signifikant geringere Verifikationszeit der Familien-basierten Strategie gegenüber der Produkt-basierten Strategie aufweisen, kann Vote Counting dennoch verwendet werden. Das Ergebnis des Vote Countings ist eine Tabelle, die alle Primärstudien inklusive ihrer Signifikanz beziehungsweise Nicht-Signifikanz auflistet.

4.1.3. Umformung der Hypothesen

Um den Wilcoxon-Vorzeichen-Rang-Test zu verwenden, muss die Untersuchung auf signifikante Beschleunigung in die Form einer Nullhypothese und einer Alternativhypothese übersetzt werden. Die formulierte Nullhypothese, die entsprechende Alternativhypothese und das für die Tests verwendete Signifikanzniveau α sind in der folgenden Auflistung formal dargestellt. Dieses Format repräsentiert gleichzeitig das Beispiel für die Formulierungen der Null- und Alternativhypothesen dieser Arbeit.

Nullhypothese $H0_0$**:** $FAM \geq PROD$

Alternativhypothese $H1_0$**:** $FAM < PROD$

Signifikanzniveau: $\alpha = 0.05\,(5\%)$

Die Nullhypothese $H0_0$ lautet in natürlicher Sprache, dass die Familien-basierte Strategie mehr oder genauso viel Zeit zur Verifikation benötigt wie die Produkt-basierte Strategie. Im Gegensatz dazu bedeutet die Alternativhypothese $H1_0$, dass die Familien-basierte Strategie signifikant schneller ist als die Produkt-basierte Strategie.
Innerhalb dieser Arbeit werden ausschließlich einseitige Hypothesen verwendet. Eine einseitige Hypothese bedeutet, dass die Alternativhypothese den Unterschied zwischen zwei Stichproben in eine Richtung spezifiziert, wie beispielsweise dass die Familien-basierte Strategie schneller ist als die Produkt-basierte Strategie. Ein Beispiel für eine zweiseitige Hypothese ist eine Alternativhypothese, die die Ungleichheit der Produkt-basierten Strategie und Familien-basierten Strategie ausdrückt, dabei weder zeigt dass die Produkt-basierte Strategie noch die Familien-basierte Strategie schneller als die jeweils andere ist. Dadurch kann geprüft werden, ob ein Unterschied besteht, jedoch nicht in welche Richtung. Das Signifikanzniveau α ist die maximale zulässige Irrtumswahrscheinlichkeit. Das Signifikanzniveau bedeutet für den Fall der Ablehnung der Nullhypothese, dass die Nullhypothese mit einer Irrtumswahrscheinlichkeit von 5% abgelehnt werden kann. Das heißt, dass mit einer Wahrscheinlichkeit von 5% die Nullhypothese fehlerhaft abgelehnt wurde, obwohl sie in Wirklichkeit wahr ist. Je kleiner das Signifikanzniveau ist, desto kleiner ist der Ablehnungsbereich für die Nullhypothese. Das bedeutet, dass die verglichenen Daten der Kontroll- und Experimentalgruppe größere Unterschiede aufweisen müssen, damit die Nullhypothese verworfen wird. Wird im Gegensatz dazu ein höheres Signifikanzniveau gewählt, steigt die Wahrscheinlichkeit für die fälschliche Ablehnung der Nullhypothese, obwohl sie in Wirklichkeit zutreffen würde. Es wird für diese Untersuchung ein Signifikanzniveau von 5% gewählt, da es sich bei der Untersuchung um keine kritische Domäne handelt. In kritischen Bereichen, wie beispielsweise bei medizinischen Experimenten, wird oftmals ein Signifikanzniveau von 1% oder 0.001% verwendet. Da die Untersuchung dieser Arbeit jedoch nur die Performanz der Verifikationsstrategien untersucht, ist ein solch geringes Signifikanzniveau nicht nötig. Um eine zu hohe Wahrscheinlichkeit für eine fälschliche Ablehnung der Nullhypothese zu vermeiden, wurde ein Signifikanzniveau von 5% gewählt, denn dieser Wert bildet einen Kompromiss zwischen der fälschlichen Ablehnung und der fälschlichen Annahme der Nullhypothese.

4.1.4. Effektgrößen

Nachdem die einzelnen Primärstudien auf Signifikanz der verglichenen Strategien untersucht wurden, wird im nächsten Schritt eine Effektgröße für die einzelnen Primärstudien bestimmt, um einen Vergleich der empirischen Ergebnisse verschiedener Primärstudien zu ermöglichen. In dieser Studie stellen Effektgrößen Werte für die Effekte dar, die durch die Verwendung einer spezifischen Verifikationsstrategie gegenüber einer anderen Verifikationsstrategie realisiert werden. Beispielsweise kann damit charakterisiert werden, welchen positiven beziehungsweise negativen Effekt die Verwendung der Familien-basierten Strategie gegenüber der Produkt-basierten Strategie erzielen kann.
Um die verschiedenen Primärstudien miteinander zu vergleichen, können verschiedene Techniken und Effektstärken verwendet werden. In dieser Arbeit wird die standardisierte Effektgröße *Cohens' d* verwendet. Eine standardisierte Effektgröße wie *Cohens' d* zeichnet sich dadurch aus, dass sie skalenfrei ist und somit besser zum Vergleich einzelner Studien verwendet werden kann. Dadurch können Unterschiede verschiedener Studien zwischen zwei Gruppen, beispielsweise der Unterschied zwischen der Verifikationszeit der Familien-basierten Strategie und der Produkt-basierten Strategie, in einem standardisierten Maß gegenübergestellt werden. Im Gegensatz dazu existieren auch unstandardisierte Effektgrößen, wie beispielsweise der Unterschied der benötigten Verifikationszeit. Zum einen ist mit diesen unstandardisierten Effektgrößen ein Vergleich der Ergebnisse mehrerer Studien schwieriger, zum anderen sind sie intuitiver, wie beispielsweise ein Ergebnis, dass die Familien-basierte Strategie in einer Primärstudie X Sekunden schneller ist als die Produkt-basierte Strategie. Die Evaluierungen der Primärstudien verwenden unterschiedlich viele Software-Produktlinien oder verschiedener Varianten derselben Software-Produktlinie. Dazu wird die standardisierte Effektgröße d_{Cohen} verwendet, um diese unterschiedlichen Evaluierungen vergleichen zu können. Denn diese Effektstärke vergleicht die Mittelwerte der unterschiedlichen Messergebnisse der Familien-basierten Strategie und der Produkt-basierten Strategie in jeder recherchierten Primärstudie.
Die Effektgröße d_{Cohen} wird nach folgender Formel 4.1 berechnet.

$$d_{Cohen} = \frac{\overline{X}_{FAM} - \overline{X}_{PROD}}{SD_{Pooled}} \tag{4.1}$$

$\overline{X}_{FAM}$ beziehungsweise $\overline{X}_{PROD}$ bezeichnet den Mittelwert pro evaluierter Software-Produktlinie der Familien-basierten beziehungsweise der Produkt-basierten Strategie. SD_{Pooled} ist die gepaarte Standardabweichung der beiden Mittelwerte, welche anhand der Formel 4.2 berechnet wird.

$$SD_{Pooled} = \sqrt{\frac{\sum((X_{FAM} - X_{PROD}) - \overline{X}_D)^2}{n-1}} \tag{4.2}$$

X_{FAM} und X_{PROD} repräsentieren die jeweiligen Verifikationszeiten für die Familien-basierte und Produkt-basierte Strategie. $\overline{X}_D$ entspricht dem Mittelwert der Differenzen der gepaarten Mittelwerte, der wie folgt berechnet wird.

$$\overline{X}_D = \frac{1}{n} * \sum(X_{FAM} - X_{PROD}) \tag{4.3}$$

Für eine ausführlichere Erläuterung zu Effektgrößen wird auf das Buch *Metaanalyse: eine anwendungsorientierte Einführung ([70])* verwiesen.

Für die Effektstärke d_{Cohen} existiert ebenfalls eine Metrik zur Interpretation. Diese Interpretationsmetrik ist in Tabelle 4.2 aufgelistet.

***d*-Wert**	**Interpretation**
$\mid 0,2 \mid - \mid 0,5 \mid$	kleiner Effekt
$\mid 0,5 \mid - \mid 0,8 \mid$	mittlerer Effekt
$\mid 0,8 \mid - \mid 1,0 \mid$	großer Effekt

Tabelle 4.2.: Interpretation des *d*-Werts nach Cohen

Der *d*-Wert kann jedoch nicht nur zwischen $+1$ und -1 liegen. Da die Interpretationsmetrik nach Cohen einen großen Effekt ab einem Betrag von $0,7$ einordnet, wird jeder Effekt der größer als 1 ist als sehr großer Effekt interpretiert. Im Gegenzug bedeutet ein Wert kleiner als $\mid 0,2 \mid$ keinen Effekt.

4.1.5. Beschreibung des Untersuchungsvorgangs

Im darauffolgenden Schritt werden die in Kapitel 3.4.2 formulierten Hypothesen untersucht. Dafür werden Grenzen für Software-Produktlinien mit wenigen und vielen Produkten, wenigen und vielen Features sowie vielen Lines of Code definiert. Zum einen werden dazu die Software-Produktlinien der Primärstudien, in denen die Familien-basierte Strategie langsamer ist als die Produkt-basierte Strategie betrachtet. Zum anderen werden die Erkenntnisse des Vote Countings und der Berechnung der Effektstärke d_{Cohen} genutzt. Somit werden für die Studien, in denen kein signifikanter Vorteil der Familien-basierten Strategie gegenüber der Produkt-basierten Strategie nachgewiesen wurde, die Anzahl der Produkte und Features der zur Evaluierung verwendeten Software-Produktlinien ermittelt. Anhand der ermittelten Menge an Produkten und Features wurden die in Tabelle 4.4 aufgelisteten Grenzen definiert. Des Weiteren enthält diese Tabelle auch die Einordnung für Software-Produktlinien mit wenigen und vielen Lines of Code.

	Wenige	***Viele***
Features	≤ 6	> 6
Produkte	≤ 8	> 8
Lines of Code	≤ 2000	> 2000

Tabelle 4.4.: Klassifizierung von Software-Produktlinien mit wenigen bzw. vielen Produkten, Features beziehungsweise Lines of Code

Anschließend an die Kategorisierung in *viele Features*, *wenige Features*, *viele Produkte*, *wenige Produkte* und *viele Lines of Code* werden die in Kapitel 3.4.2 vorgestellten Hypothesen untersucht. Der Untersuchungsvorgang für die Hypothesen ist in der folgenden Aufzählung dargestellt.

1. Formulierung der Nullhypothese und der Alternativhypothese für die jeweilige untersuchte Hypothese.

2. Mittelwert-Bildung der Verifikationszeit für Primärstudien, in denen die jeweilige Strategie in Kombination mit dem Model Checking Ansatz für mehrere logische Formeln evaluiert wurde. Dieser Schritt wird nur für Primärstudien bezüglich Model Checking durchgeführt.
3. Aggregierung der Daten der jeweiligen Klassifizierung für Produkte, Features und Lines of Code der einzelnen Primärstudien in tabellarischer Form.
4. Untersuchung der Hypothesen (in Form der Nullhypothese und Alternativhypothese) mit dem Wilcoxon-Vorzeichen-Rang-Test.
5. Darstellung der aggregierten Daten für die gegenübergestellten Verifikationsstrategien in ein gemeinsames Meta-Diagramm. Meta-Diagramme stellen in dieser Studie die Verifikationszeit oder den Speicherverbrauch zweier Strategien in Bezug auf die unabhängigen Variablen dar.
6. Interpretation der Meta-Diagramme und Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests.

Dieser Untersuchungsvorgang kann für die Bereiche Static Analysis und Theorem Proving nicht verwendet werden. Dazu ist die Menge an empirischen Messdaten zu gering, um eine Untersuchung der Hypothesen anhand des Signifikanztests durchzuführen. Für diese beiden Analysetechniken wird ausschließlich anhand der gesamten Evaluierungsergebnisse der jeweiligen Primärstudien Meta-Diagramme erstellt und anschließend der erkennbare Trend dieser Diagramme interpretiert.

Die statistischen Methoden und Hypothesentests werden in dieser Arbeit mit der Statistiksoftware *R* [1] ausgeführt. Die Statistiksoftware *R* wird aus mehreren Gründen ausgewählt. *R* bietet eine gute Dokumentation und eine Hilfefunktion für die jeweiligen Befehle für die statistischen Tests. Außerdem bietet *R* eine einfache Möglichkeit, Datensätze aus Tabellen einzulesen und vorliegende Daten in verschiedenen Diagrammen darzustellen. *R* bietet standardmäßig eine integrierte Konsoleneingabe, es existieren dennoch auch verschiedene, zusätzlich installierbare graphische Oberflächen zur Verwendung von *R*. Ein weiterer Vorteil von *R* ist die Existenz von zahlreichen zusätzlichen Bibliotheken, wie beispielsweise für die Durchführung einer Meta-Analyse und zum Erzeugen von *Forest Plots*. Diese Forest Plots werden beispielsweise für die Darstellung der Effektstärke d_{Cohen} der einzelnen Primärstudien benötigt.
Der Wilcoxon-Vorzeichen-Rang-Test wird somit ebenfalls mit der Statistiksoftware *R* ausgeführt. Der allgemeine entsprechende *R*-Befehl für diesen Test ist nachfolgend dargestellt, wobei *wertLinkeSeite* und *wertRechteSeite* den Wert auf der linken beziehungsweise rechten Seite der Nullhypothese bezeichnen. Die Attribute *paired=TRUE* und *alternative="less"* spezifizieren eine abhängige Stichprobe sowie eine einseitige Hypothese, wobei *less* bedeutet, dass der Wert auf der linken Seite der Alternativhypothese kleiner ist als der Wert auf der rechten Seite der Alternativhypothese.

wilcox.test(wertLinkeSeite,wertRechteSeite,paired=TRUE,alternative="less")

Fortan werden in dieser Arbeit verschiedene Abkürzungen verwendet, welche in der Tabelle 4.6 dargestellt sind.

[1] https://www.r-project.org/

Abkürzung	Bedeutung
PROD	Abkürzung für die Produkt-basierte Strategie
PROD-OPT	Abkürzung für die optimierte Produkt-basierte Strategie
$PROD_{PARA}$	Abkürzung für die parallele Produkt-basierte Strategie
$PROD_{SEQ}$	Abkürzung für die sequenzielle Produkt-basierte Strategie
FAM	Abkürzung für die Familien-basierte Strategie
FEAT	Abkürzung für die Feature-basierte Strategie
FEAT-PROD	Abkürzung für die Feature-Produkt-basierte Strategie
SAMPLE	Abkürzung für die Sample-basierte Strategie (optimierte Produkt-basierte Strategie)
Speed Up	Abkürzung für die Beschleunigung durch Verwendung einer Verifikationsstrategie verglichen mit einer anderen Verifikationsstrategie
# Zustände	Anzahl der Zustände, die beim Model Checking exploriert werden

Tabelle 4.6.: Abkürzungen für die Untersuchung der Hypothesen

In den nachfolgenden Abschnitten werden die formulierten Hypothesen anhand der in diesem Kapitel beschriebenen Methoden untersucht.

4.2. Untersuchung der Hypothesen bezüglich Model-Checking

In diesem Abschnitt werden die in Abschnitt 3.4.2 formulierten Hypothesen bezüglich der Vorteile bestimmter Strategien zum Model Checking von Software-Produktlinien mit spezifischen Eigenschaften untersucht. Am Anfang dieses Abschnitts werden die verwendeten Primärstudien für das Model Checking von Software-Produktlinien zusammengefasst. Danach wird das allgemeine Verhältnis der Familien-basierten und Produkt-basierten Strategie in Abhängigkeit der betrachteten abhängigen Variablen in Form von Meta-Diagrammen erläutert. Daraufhin werden die einzelnen Model-Checking-Primärstudien, die Evaluierungen zu verschiedenen Strategien enthalten, auf Signifikanz untersucht. Des Weiteren wird für diese Primärstudien eine standardisierte Effektgröße für die Verifikationszeit der Produkt-basierten Strategie gegenüber der Familien-basierten Strategie berechnet. Anschließend erfolgt die Untersuchung der formulierten Hypothesen mittels Hypothesentests.

4.2.1. Zusammenfassung der Primärstudien

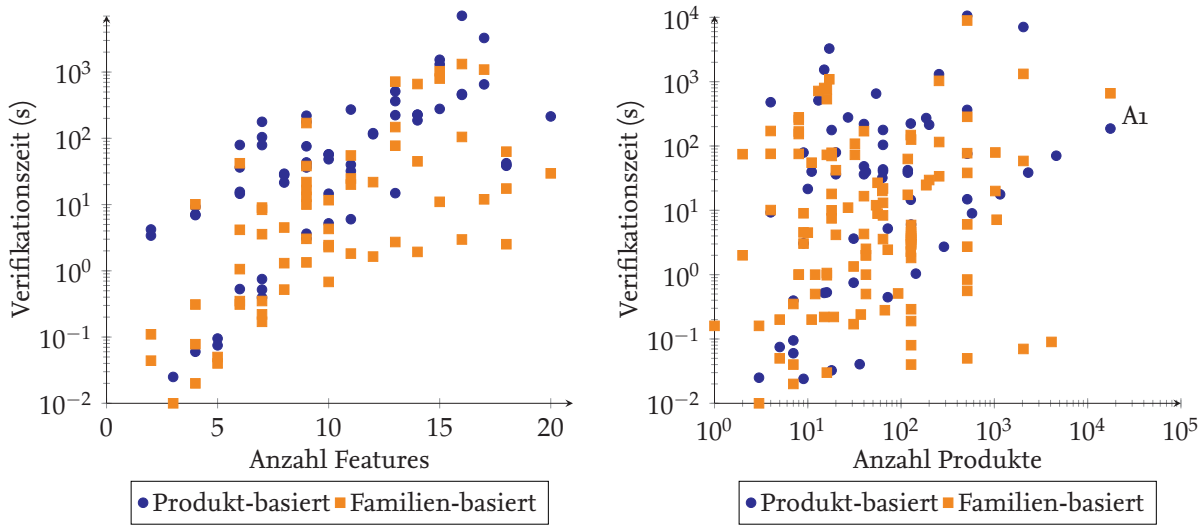
Die in Tabelle 4.6 aufgelisteten Abkürzungen werden unter anderem auch in der Tabelle A.1 im Anhang A in Kapitel A.1.1 verwendet. In der Tabelle A.1 sind die 28 Primärstudien dargestellt, die sich auf Ansätze zum Model Checking von Software-Produktlinien beziehen. In der Tabelle befindet sich in der ersten Spalte namens *Dokument* das Kürzel des Dokuments (siehe Kapitel 3.3.1), in der die jeweiligen Evaluierungsergebnisse enthalten sind. Die mit dem Namen *Evaluierte SPLs* bezeichnete Spalte listet die Software-Produktlinien inklusive der unabhängigen Variablen in Form der Anzahl Features, Produkte und Codezeilen auf. Die dritte Spalte zeigt die abhängigen Variablen, welche im entsprechenden Dokument gemessen wurden. Die vierte Spalte stellt die Verifikationsstrategien dar, für welche der jeweilige Ansatz zum Model Checking evaluiert wurde.

Die Tabelle A.1 zeigt für die 28 Primärstudien zusammenfassend, dass in 24 Studien die Verifikationszeit der betrachteten Strategien evaluiert wurde. In fünf Primärstudien wurde der Speicherverbrauch in der Anzahl der Zustände gemessen [29, 7, 8, 30, 31], in vier Studien wurde der Speicherverbrauch in Megabytes oder Bytes gemessen [26, 28, 25, 2]. Zusätzlich wurde in drei der 28 Primärstudien der Speed Up ermittelt, der aus dem Vergleich der Verifikationszeit einer Strategie mit einer anderen Strategie resultiert [40, 58, 56]. Des Weiteren unterscheiden sich die Primärstudien darin, welche Verifikationsstrategien sie mit der für ihren neu entwickelten Ansatz genutzten Strategie vergleichen. Eine ausschließliche Betrachtung der Familien-basierten Strategie erfolgt in sechs Primärstudien [58, 32, 31, 2, 68, 78], wohingegen eine alleinige Betrachtung der Produkt-basierten Strategie [75] und der Feature-basierten Strategie [45] in jeweils einer Primärstudie erfolgt. Zusätzlich wird in einer Primärstudie [38] die Feature-Produkt-basierte Strategie mit der Produkt-basierten Strategie verglichen. Außerdem wird in der Primärstudie [15] sowohl eine unoptimierte als auch eine optimierte Produkt-basierte Strategie evaluiert. Die restlichen 18 Studien vergleichen die Familien-basierte Strategie mit der Produkt-basierten Strategie.
Die zur Evaluierung verwendeten Software-Produktlinien in den Primärstudien unterscheiden sich in der Anzahl der Features und der Anzahl der Produkte. Die Anzahl an Features liegt zwischen zwei [30, 2, 78] und 4675 Features [32], wobei die meisten Software-Produktlinien zwischen zwei und 20 Features besitzen. Die Anzahl der Produkte der verwendeten Software-Produktlinien befindet sich in einem Bereich von zwei [29, 78] bis 17544 Produkten [25], die meisten betrachteten Software-Produktlinien enthalten zwischen acht und 256 Produkte. Nur in einer der recherchierten Primärstudien wird die Anzahl der Quellcode-Zeilen dokumentiert [71].
Außerdem listet die Tabelle detailliert auf, welche Primärstudie welche Anzahl an Software-Produktlinien oder welche Variationen der verwendeten Software-Produktlinien für die Evaluierung verwendet wurden. In neun Primärstudien wurden mehrere unterschiedliche Software-Produktlinien genutzt [58, 32, 71, 30, 31, 56, 50, 45, 15], in sechs Primärstudien wurden verschiedene Variationen der selben Software-Produktlinie für die Evaluierung eingesetzt [33, 29, 57, 10, 2, 78]. In zwei Primärstudien wurden mehrere Software-Produktlinien in unterschiedlichen Ausprägungen herangezogen [40, 6]. Die genauen Eigenschaften der recherchierten Primärstudien sind in Tabelle A.1 im Anhang A zu finden.

4.2.2. Allgemeine Meta-Diagramme

Das Diagramm 4.1a zeigt das Streudiagramm für die Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten in Abhängigkeit der Anzahl der Features. Das Diagramm zeigt einen deutlichen Trend, dass die Familien-basierte Strategie schneller ist als die Produkt-basierte Strategie. Außerdem ist ersichtlich, dass die Verifikationszeit beider Strategien mit zunehmender Anzahl der Features ansteigt.
Das Streudiagramm 4.1b zeigt denselben Vergleich in Abhängigkeit der Anzahl der Produkte. In Diagramm 4.1b ist nicht so prägnant erkennbar, dass die Familien-basierte Strategie schneller ist als die Produkt-basierte Strategie. Jedoch ist dieser Trend auch in diesem Diagramm teilweise ersichtlich. Wie erwartet ist ebenfalls zu erkennen, dass die Verifikationszeit beider Strategien mit zunehmender Produktanzahl ansteigt. In dem Streudiagramm 4.1b sind auch die Datenpaare besser erkennbar (insgesamt 5, siehe Tabelle A.8), bei denen die Produkt-basierte Strategie schneller ist als die Familien-basierte Strategie. Ein Beispiel dafür ist das Datenpaar, welches mit *A1* annotiert

ist, bei dem die Produkt-basierte Strategie schneller ist als die Familien-basierte Strategie.



(a) Zeit vs. Features

(b) Zeit vs. Produkte

Abbildung 4.1.: Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit (a) der Anzahl der Features und (b) der Anzahl der Produkte

4.2.3. Vote Counting der Primärstudien

Da für den Wilcoxon-Vorzeichen-Rang-Test die Stichprobengröße mindestens gleich fünf sein muss, konnten nicht alle 28 Primärstudien bezüglich Model Checking werden. Insgesamt konnten dabei nur 18 Evaluierungen aus 9 dieser Primärstudien, welche einen Vergleich der Familien-basierten Strategie mit der Produkt-basierten Strategie enthalten, genutzt werden. Es konnten nur Primärstudien herangezogen werden, die entweder während der Evaluierung mehrere Eigenschaften für das Model Checking genutzt haben, zur Evaluierung mehrere Software-Produktlinien verwendet haben oder die evaluierte Software-Produktlinie in Variationen unterschiedlicher Anzahl Features und Produkte vorlag.

Das Ergebnis der Signifikanz-Untersuchung der einzelnen Primärstudien ist in der Tabelle 4.8 dargestellt. In der ersten Spalte der Tabelle ist das Dokument aufgelistet, aus welchem der Datensatz für den Signifikanztest entnommen wurde. Jede Zeile der Tabelle entspricht je einer zur Evaluierung verwendeten Software-Produktlinie. Mit einem *-Symbol markierte Dokumentenkürzel bezeichnen Software-Produktlinien, die zur Evaluierung in unterschiedlichen Ausprägungen in Form verschiedener Anzahl Features und Produkten genutzt wurden. Kursive Dokumentenkürzel stellen Evaluierungen dar, bei denen unterschiedliche Eigenschaftsspezifikationen in Form mehrerer logischer Formeln für die Evaluierung verwendet wurden. Die Spalte mit der Bezeichnung w beinhaltet den Prüfwert, der durch den Wilcoxon-Vorzeichen-Rang-Test berechnet wird. Mit n wird der Stichprobenumfang bezeichnet. Der Wert $w_{\alpha,n}$ ist der kritische Wert des Wilcoxon-Vorzeichen-Rang-Tests.

Dokument	w	n	$w_{\alpha,n}$	**Nullhypothese abgelehnt, wenn:** $w \leq w_{\alpha,n}$	p**-Wert**	**Ablehnung anhand des** p**-Werts**
MC-A 2*	0	8	5	$0 \leq 5 \rightarrow$ Ja	0,00391	Ja
MC-A 2*	0	5	0	$0 \leq 0 \rightarrow$ Ja	0,0313	Ja
MC-A 2*	9	7	3	$9 \leq 3 \rightarrow$ Nein	0,234	Nein
MC-A 3	0	10	10	$0 \leq 10 \rightarrow$ Ja	0,000977	Ja
MC-A 7	0	15	30	$0 \leq 30 \rightarrow$ Ja	0,0000305	Ja
MC-A 7	0	15	30	$0 \leq 30 \rightarrow$ Ja	0,0000305	Ja
MC-A 7	15	15	30	$15 \leq 30 \rightarrow$ Ja	0,00418	Ja
MC-A 7	8	15	30	$8 \leq 30 \rightarrow$ Ja	0,00763	Ja
MC-A 8	0	45	371	$0 \leq 371 \rightarrow$ Ja	$\sim 2{,}677 * 10^{-9}$	Ja
MC-A 10	0	9	8	$0 \leq 8 \rightarrow$ Ja	0,00195	Ja
MC-A 10	9	10	10	$9 \leq 10 \rightarrow$ Ja	0,0322	Ja
MC-A 10	0	9	8	$0 \leq 8 \rightarrow$ Ja	0,00195	Ja
MC-A 10	0	6	2	$0 \leq 2 \rightarrow$ Ja	0,0156	Ja
MC-A 10	0	5	0	$0 \leq 0 \rightarrow$ Ja	0,0313	Ja
MC-A 10	0	5	0	$0 \leq 0 \rightarrow$ Ja	0,0313	Ja
MC-A 11	78	12	17	$78 \leq 17 \rightarrow$ Nein	~0,999	Nein
MC-A 11	7	12	17	$7 \leq 17 \rightarrow$ Ja	0,00464	Ja
MC-A 11	0	12	17	$0 \leq 17 \rightarrow$ Ja	0,000244	Ja
MC-A 12	6	6	2	$6 \leq 2 \rightarrow$ Nein	0,219	Nein
MC-A 12	0	6	2	$0 \leq 2 \rightarrow$ Ja	0,0156	Ja
MC-A 17	70	12	17	$70 \leq 17 \rightarrow$ Nein	0,995	Nein
MC-A 17	2	12	17	$2 \leq 17 \rightarrow$ Ja	0,000723	Ja
MC-A 20*	0	9	8	$0 \leq 8 \rightarrow$ Ja	0,00195	Ja
MC-A 20*	0	8	5	$0 \leq 5 \rightarrow$ Ja	0,00391	Ja
MC-A 20*	0	8	5	$0 \leq 5 \rightarrow$ Ja	0,00391	Ja

Tabelle 4.8.: Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die einzelnen Primärstudien in Bezug auf signifikanter Verbesserung der Verifikationszeit der Familien-basierten Strategie verglichen mit der Produkt-basierten Strategie

Die Nullhypothese wird bei dem Wilcoxon-Vorzeichen-Rang-Test abgelehnt wenn $w \leq w_{\alpha}, n$ ist. Wenn die Nullhypothese abgelehnt wird, ist das in der Tabelle mit *Ja* gekennzeichnet. Das Verwerfen der Nullhypothese bedeutet, dass die Alternativhypothese angenommen wird und somit die Familien-basierte Strategie in der entsprechenden Studie signifikant schneller ist als die Produkt-basierte Strategie. Beispielsweise wird in der ersten Zeile der Tabelle die Nullhypothese verworfen. Wenn die Nullhypothese anhand des Signifikanztests nicht verworfen werden kann, enthält die Tabellenspalte ein *Nein*, wie unter anderem in der dritten Zeile der Tabelle. Nicht verwerfen der Nullhypothese bedeutet, dass in der spezifischen Studie die Familien-basierte Strategie nicht signifikant schneller ist als die Produkt-basierte Strategie. Dennoch bedeutet es im Gegenzug nicht, dass die Produkt-basierte Strategie schneller ist als die Familien-basierte Strategie, sondern es kann aufgrund der verfügbaren Daten keine Aussage bezüglich des signifikanten Vorteils der Familien-

basierten Strategie getroffen werden. Die Tabelle 4.8 enthält zusätzlich noch den p-Wert, der ebenfalls beim Wilcoxon-Vorzeichen-Rang-Test mit R berechnet wird. p-Werte, die mit einer Tilde beginnen, sind gerundete Werte. Ist der p-Wert kleiner als das Signifikanzniveau α, deutet diese Tatsache ebenfalls auf eine Ablehnung der Nullhypothese hin. Für eine ausführlichere Erläuterung des p-Werts wird auf „Hedderich und Sachs"[44] verwiesen. Da in dieser Untersuchung hauptsächlich das Ablehnungskriterium des Wilcoxon-Vorzeichen-Rang-Tests angewendet wird und der p-Wert nur zur zusätzlichen Bestätigung genutzt wird, sind gerundete p-Werte ausreichend.
Die vorherige Tabelle 4.8 stellt das *Vote Counting* bezüglich der signifikanten und nicht signifikanten Vergleiche der Familien-basierten Strategie und der Produkt-basierten Strategie der einzelnen Primärstudien dar. Die Tabelle 4.8 zeigt, dass bei 21 der 25 betrachteten einzelnen Evaluierungsergebnisse die Familien-basierte Strategie signifikant schneller ist als die Produkt-basierte Strategie. Nur bei vier Studien konnte diese signifikante Verbesserung durch die Familien-basierte Strategie nicht gezeigt werden. Somit ist die Familien-basierte Strategie bei 84% der betrachteten Evaluierungen signifikant schneller. Dieses Ergebnis zeigt einen generellen Vorteil der Familien-basierten Strategie ohne Betrachtung der unabhängigen Variablen wie Anzahl der Produkte oder Features. Dennoch konnte für vier der verwendeten Evaluierungsergebnisse kein signifikanter Vorteil der Familien-basierten Strategie gegenüber der Produkt-basierten Strategie gezeigt werden.

Des Weiteren werden die Ansätze der weiteren möglichen Verifikationsstrategien wie Sampling-basierten oder Familien-basierten Strategien inklusive Abstraktionstechniken auf Signifikanz untersucht. Diese Ergebnisse sind in den folgenden Tabellen 4.10 und 4.12 dargestellt. Die Tabelle 4.10 zeigt die Ergebnisse des Signifikanztests bezüglich der in der Primärstudie von Apel et al. [71] verglichenen Strategien. Dabei wurde der Test immer darauf überprüft, ob Strategie A signifikant schneller ist als Strategie B. Jede der in der Primärstudie verwendeten Software-Produktlinien besitzt in der Tabelle 4.10 eine eigene Spalte, wobei die Software-Produktlinie *Minepump* durch *MP* abgekürzt wurde. Das ✓-Symbol kennzeichnet dabei für jedes überprüfte Strategie-Paar und jede Software-Produktlinie dass die Strategie A signifikant schneller ist als die Strategie B. Ist die Strategie A anhand des Wilcoxon-Vorzeichen-Rang-Tests nicht signifikant schneller, ist dieser Fall analog durch das ✗-Symbol in der entsprechenden Zelle markiert. Zusammenfassend zeigt sich, dass alle drei Sampling-Strategien signifikant schneller sind als die Produkt-basierte Strategie. Beim Vergleich der Familien-basierten Strategie und den verschiedenen Sampling-Strategien stellt sich ein gemischtes Ergebnis dar. Teilweise ist die Familien-basierte Strategie signifikant schneller, teilweise nicht. In einem Fall ist sogar das Single-Wise Sampling signifikant schneller als der Familien-basierte Ansatz. Eine detaillierte Darstellung des durchgeführten Wilcoxon-Vorzeichen-Rang-Tests ist im Anhang A in der Tabelle A.5 einsehbar.

Strategie A	Strategie B	E-Mail Java	E-Mail C	Elevator Java	Elevator C	MP Java	MP C
SINGLE-WISE	PROD	✓	✓	✓	✓	✓	✓
PAIR-WISE	PROD	✓	✓	✓	✓	✓	✓
TRIPLE-WISE	PROD	✓	✓	✓	✓	✓	✓
FAM	SINGLE-WISE	✗	✗	✗	✓	✓	✓
FAM	PAIR-WISE	✗	✗	✓	✓	✓	✓
FAM	TRIPLE-WISE	✓	✗	✓	✓	✓	✓
SINGLE-WISE	FAM	✗	✓	✗	✗	✗	✗
SINGLE-WISE	FAM	✗	✗	✗	✗	✗	✗
SINGLE-WISE	FAM	✗	✗	✗	✗	✗	✗

Tabelle 4.10.: Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die miteinander verglichenen Verifikationsstrategien in der Primärstudie MC-A 10 bezüglich der Verifikationszeit

Auch in der Tabelle 4.12 werden die Ergebnisse der Signifikanztests der Verifikationszeit weiterer Verifikationsstrategien der Model-Checking-Primärstudien dargestellt. In den Primärstudien von Sabouri und Hamideh [31] sowie von Dimovski et al. [2] wurden Abstraktionstechniken für die Familien-basierte Strategie vorgestellt und mit der einfachen Familien-basierten Strategie verglichen. Die Primärstudie von Ghezzi und Sharifloo [15] bietet einen optimierten Produkt-basierten Ansatz und evaluiert diesen Ansatz im Vergleich zu einem unoptimierten Produkt-basierten Ansatz. Bei den Signifikanztests wurde überprüft, ob die jeweilige in der Tabelle 4.12 dargestellte Strategie A signifikant schneller ist als die Strategie B. Bestätigt der Test diese Untersuchung, ist in der letzten Spalte der Tabelle das ✓-Symbol enthalten. In allen drei Studien sind die Strategien inklusive der Optimierungsverfahren signifikant schneller als die entsprechenden herkömmlichen Strategien.

Dokument	Strategie A	Strategie B	w	$w_{\alpha,n}$	Hypothese bestätigt?
MC-A 4	FAM-ABSTRACT	FAM	0	10	✓
MC-A 23	FAM-ABSTRACT	FAM	0	2	✓
MC-A 26	PROD-OPT	PROD	6	10	✓

Tabelle 4.12.: Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die weiteren verglichenen Verifikationsstrategien in den Primärstudien zum Model Checking bezüglich der Verifikationszeit

Außerdem wurde auch der Speicherverbrauch der verschiedenen Verifikationsstrategien auf signifikante Unterschiede überprüft. Die Ergebnisse dieses Wilcoxon-Vorzeichen-Rang-Tests sind in der Tabelle 4.14 aufgelistet. Die Tabelle 4.14 hat dieselbe Struktur wie die Tabelle 4.12, mit der Ausnahme der Spalte *Anmerkung*. Auch in dieser Untersuchung wurde überprüft, ob die Strategie A einen signifikant geringeren Speicherverbrauch besitzt als die Strategie B. In der Tabelle 4.14 ist zu erkennen, dass in fast allen Primärstudien mit Evaluierungen des Speicherverbrauchs die Familien-basierte Strategie mit der Produkt-basierten Strategie verglichen wurde. In all diesen Fällen weist die Familien-basierte Strategie einen signifikant geringeren Speicherverbrauch auf als die Produkt-basierte Strategie. Weiterhin zeit die Tabelle 4.14, dass die Familien-basierte Strategie inklusive Abstraktionsmethoden aus der Primärstudie von Lochau et al. [50] signifikant weniger Speicher verbraucht als die reine Familien-basierte Strategie.

Dokument	Anmerkung	Strategie A	Strategie B	w	$w_{\alpha,n}$	Hypothese bestätigt?
MC-A 5	-	FAM	PROD	0	2	✓
MC-A 8	-	FAM	PROD	0	371	✓
MC-A 13	Ohne Koordinator	FAM	PROD	0	17	✓
MC-A 13	Mit Koordinator	FAM	PROD	7	8	✓
MC-A 20	-	FAM	PROD	0	8	✓
MC-A 24	-	FAM-ABSTRACT	FAM	0	10	✓

Tabelle 4.14.: Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für den Speicherverbrauch der in den Primärstudien verglichenen Verifikationsstrategien zum Model Checking

4.2.4. Cohens' d zum Vergleich der Primärstudien

In der folgenden Abbildung 4.2 sind die nach Cohen berechneten d-Werte der einzelnen Primärstudien bezüglich der Verifikationszeit dargestellt. Die Punkte jeder Studie stellen die berechnete Effektgröße d_{Cohen} dar. Die horizontalen Linien der jeweiligen Studie visualisieren das 95%-Konfidenzintervall für die berechnete Effektgröße. Die Tabelle A.7 im Anhang A zeigt alle Werte und Ergebnisse für die Berechnung der Effektgröße d_{Cohen}.

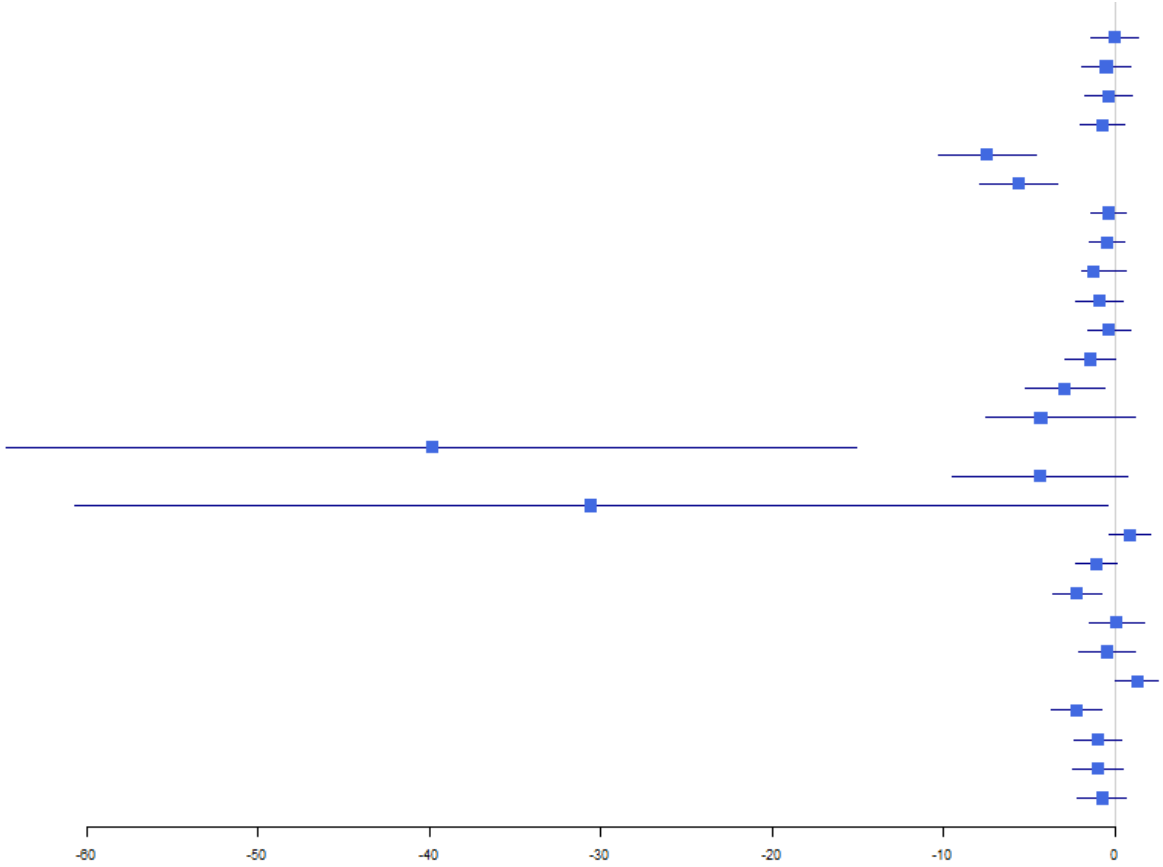


Abbildung 4.2.: Effektgröße Cohen's d der einzelnen Primärstudien bezüglich der Verifikationszeit

52 % der betrachteten Evaluierungen haben einen d-Wert zwischen $-0,3$ und $-2,0$. Ein negativer d-Wert bedeutet in diesem Fall, dass die Familien-basierte Strategie eine geringere Verifikationszeit in der entsprechenden Studie aufweist als die Produkt-basierte Strategie. 26 % der berechneten Effektgrößen d_{Cohen} liegen zwischen $-2,0$ und $-5,0$. Die 15. und 17. der untersuchten Studien besitzen einen d-Wert von $-39,9$ beziehungsweise $-30,6$. Allerdings sind in Abbildung 4.2 auch Ausreißer in die entgegengesetzte Richtung erkennbar. Diese Ausreißer sind die 18., die 21. und 23. der untersuchten Studien. Hier wurde ein positiver d-Wert berechnet, bei der 18. Studie ein Wert von $0,818$ und bei der 23. sogar ein Wert von $1,26$. Bei diesen drei Ausreißern ist die Produkt-basierte Strategie schneller als die Familien-basierte Strategie, bei zwei der drei Fälle weist die Produkt-basierte Strategie einen großen beziehungsweise sehr großen Effekt gegenüber der Familien-basierten Strategie auf.

Die Tabelle 4.17 fasst die berechneten standardisierten Effektgrößen der betrachteten Primärstudien zusammen. In der ersten Spalte der Tabelle ist die Intervallbreite der jeweiligen Effektklasse aufgeführt und die zweite Spalte enthält die Bedeutung dieses Intervalls. In der dritten Spalte ist die Anzahl der Effektgröße der jeweiligen Effektklasse dargestellt inklusive der entsprechenden Primärstudien. Primärstudien, die mehrere Software-Produktlinien zur Evaluierung verwenden und dadurch für jede dieser Software-Produktlinien die Effektgröße berechnet wurde, sind für jede zutreffende Effektklasse nur einmal referenziert. Die genauen Ergebnisse sind in der Tabelle A.7 im

Anhang A zu finden.

d-Wert Intervall	Bedeutung	Anzahl
$(0,0\,;\,2,0]$	negativer Effekt FAM	*3* [57, 10, 75]
$(-0,2\,;\,0,0]$	kein Effekt FAM	*1* [40]
$(-0,5\,;\,-0,2]$	kleiner positiver Effekt FAM	*4* [40, 7, 71, 10]
$(-0,8\,;\,-0,5]$	mittlerer positiver Effekt FAM	*4* [40, 55, 7, 6]
$[-1,0\,;\,-0,8]$	großer positiver Effekt FAM	*1* [71]
$(-\infty\,;\,-1,0)$	sehr großer positiver Effekt FAM	*14* [7, 8, 71, 57, 75, 28, 6]

Tabelle 4.17.: Zusammenfassung der d_{Cohen}-Werte der einzelnen Primärstudien

4.2.5. Untersuchte Hypothesen

Im Abschnitt 4.1 wurde bereits die Formulierung einer Nullhypothese und einer Alternativhypothese erläutert. Die Tabelle 4.19 zeigt die Nullhypothese, die Alternativhypothese und das Signifikanzniveau für die jeweilige Hypothese bezüglich der Performanz der verglichenen Verifikationsstrategien.

Hypothese	Nullhypothese	Alternativhypothese	Signifikanzniveau
GH1	$H0_{MC1} : FAM \geq PROD$	$H1_{MC1} : FAM < PROD$	$\alpha = 0,05$
GH2	$H0_{MC2} : PROD \geq FAM$	$H1_{MC2} : PROD < FAM$	$\alpha = 0,05$
GH3	$H0_{MC3} : FAM \geq PROD$	$H1_{MC3} : FAM < PROD$	$\alpha = 0,05$
GH4	$H0_{MC4} : PROD \geq FAM$	$H1_{MC4} : PROD < FAM$	$\alpha = 0,05$
GH5	$H0_{MC5} : FAM \geq PROD$	$H1_{MC5} : FAM < PROD$	$\alpha = 0,05$
GH6	$H0_{MC6} : FAM \geq PROD_{PARA}$	$H1_{MC6} : FAM < PROD_{PARA}$	$\alpha = 0,05$
GH7	$H0_{MC7} : PROD_{SEQ} \geq FAM$	$H1_{MC7} : PROD_{SEQ} < FAM$	$\alpha = 0,05$

Tabelle 4.19.: Nullhypothesen, Alternativhypothesen und das Signifikanzniveau für die Hypothesen bezüglich Model Checking

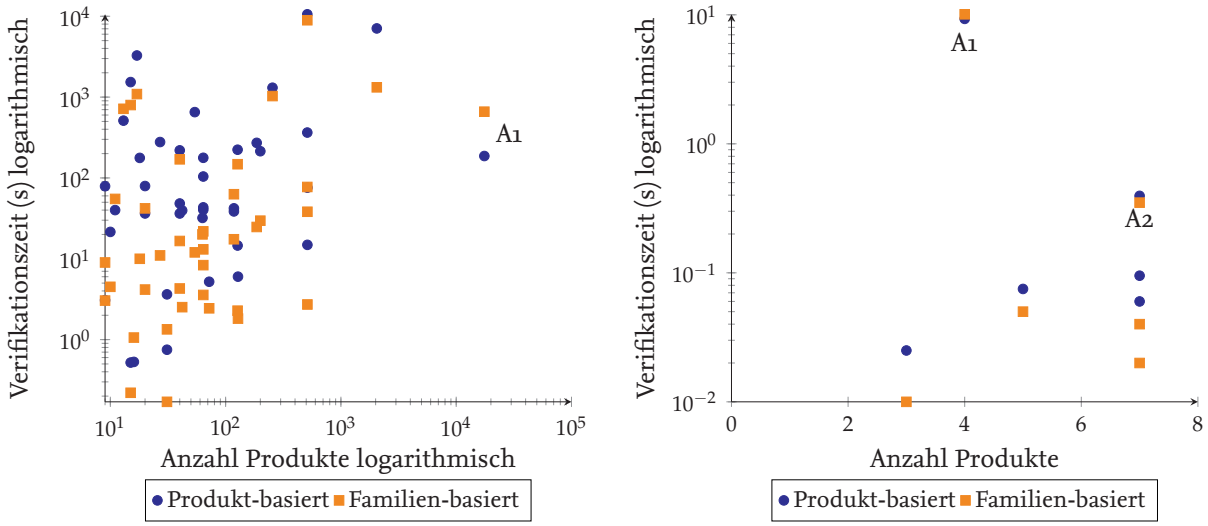
Die gebildeten Werte und die Aggregation dieser Mittelwerte sind in den Tabellen A.8 - A.14 in Anhang A einzusehen.
Die entsprechenden *R*-Befehle für die durchgeführten Wilcoxon-Vorzeichen-Rang-Tests für die Untersuchung der Hypothesen bezüglich Model Checking sind in der Tabelle A.16 in Anhang A zu finden. Die Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests sind in der Tabelle 4.21 dargestellt.

Null-hypothese	w	n	$w_{\alpha,n}$	Nullhypothese abgelehnt, wenn $w \leq w_{\alpha,n}$	p-Wert	Ablehnung anhand des p-Werts
$H0_{MC1}$	93	40	286	$93 \leq 286 \rightarrow$ Ja	$2{,}26 * 10^{-6}$	Ja
$H0_{MC2}$	15	6	2	$15 \leq 2 \rightarrow$ Nein	0,844	Nein
$H0_{MC3}$	103	48	426	$103 \leq 426 \rightarrow$ Ja	$2{,}28 * 10^{-8}$	Ja
$H0_{MC4}$	94	14	25	$94 \leq 25 \rightarrow$ Nein	0,9974	Nein
$H0_{MC5}$	-	-	-	-	-	-
$H0_{MC6}$	0	11	13	$0 \leq 13 \rightarrow$ Ja	0,000488	Ja
$H0_{MC7}$	0	11	13	$0 \leq 13 \rightarrow$ Ja	0,000488	Ja

Tabelle 4.21.: Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die Hypothesen bezüglich Model Checking

4.2.6. Meta-Diagramme der Hypothesen

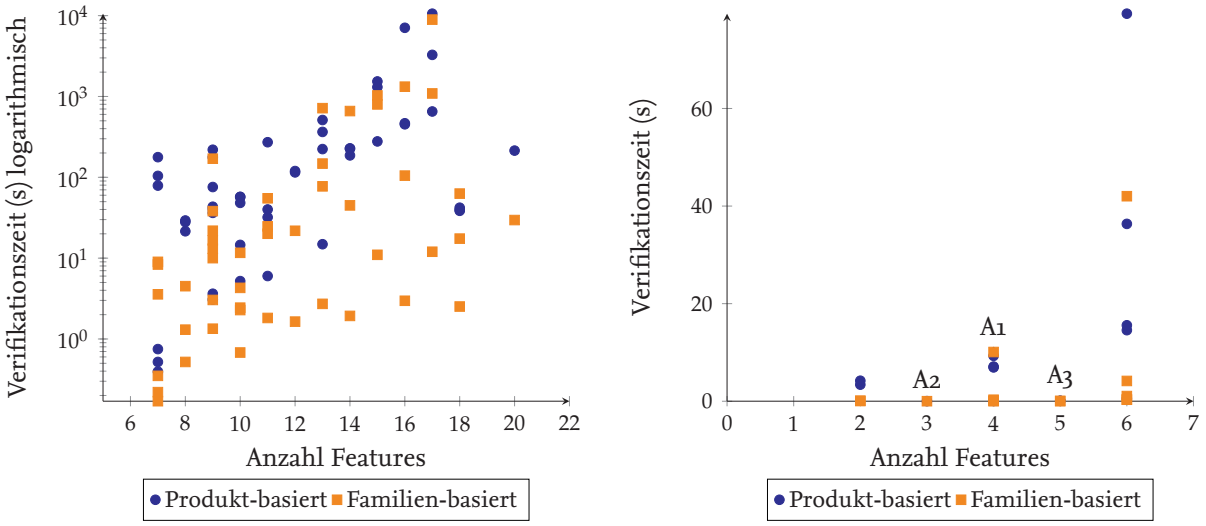
Dieser Abschnitt enthält die Abbildungen 4.3a bis 4.6b mit den Meta-Diagrammen der Kategorien weniger und vieler unabhängiger Variablen und den zugehörigen abhängigen Variablen der verglichenen Verifikationsstrategien. Beispielsweise zeigt Abbildung 4.3a die Verifikationszeit der Produkt-basierten und Familien-basierten Strategie für viele Produkte für die Hypothese GH1, wohingegen Abbildung 4.3b die Verifikationszeit dieser Strategien für wenige Produkte für die Hypothese GH2 darstellt.



(a) Zeit FAM & PROD (b) Zeit FAM & PROD

Abbildung 4.3.: Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien mit (a) vielen Produkten und (b) wenigen Produkten

Die Abbildungen 4.4a und 4.4b repräsentieren die Verifikationszeit der Produkt-basierten und Familien-basierten Strategie für viele beziehungsweise wenige Features für die Hypothesen GH3 beziehungsweise GH4.



(a) Zeit FAM & PROD (b) Zeit FAM & PROD

Abbildung 4.4.: Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien mit (a) vielen Features und (b) wenigen Features

Die Abbildung 4.5 dient zur Darstellung der Daten für die Untersuchung der Hypothese GH5 und zeigt somit die Verifikationszeit der Produkt-basierten Strategie und Familien-basierten Strategie in Bezug auf die Anzahl der Codezeilen der evaluierten Software-Produktlinien.

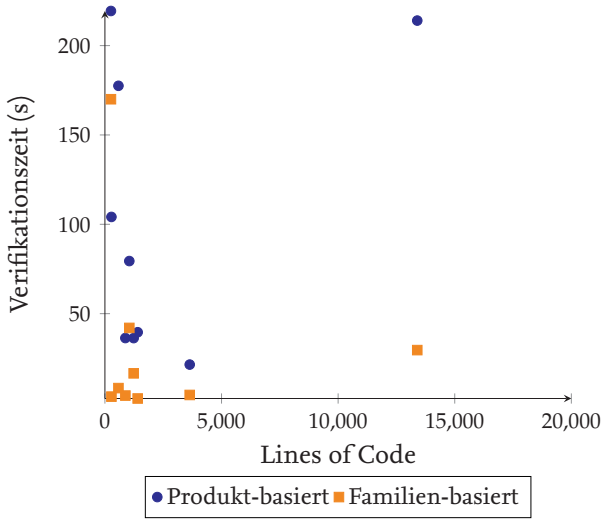
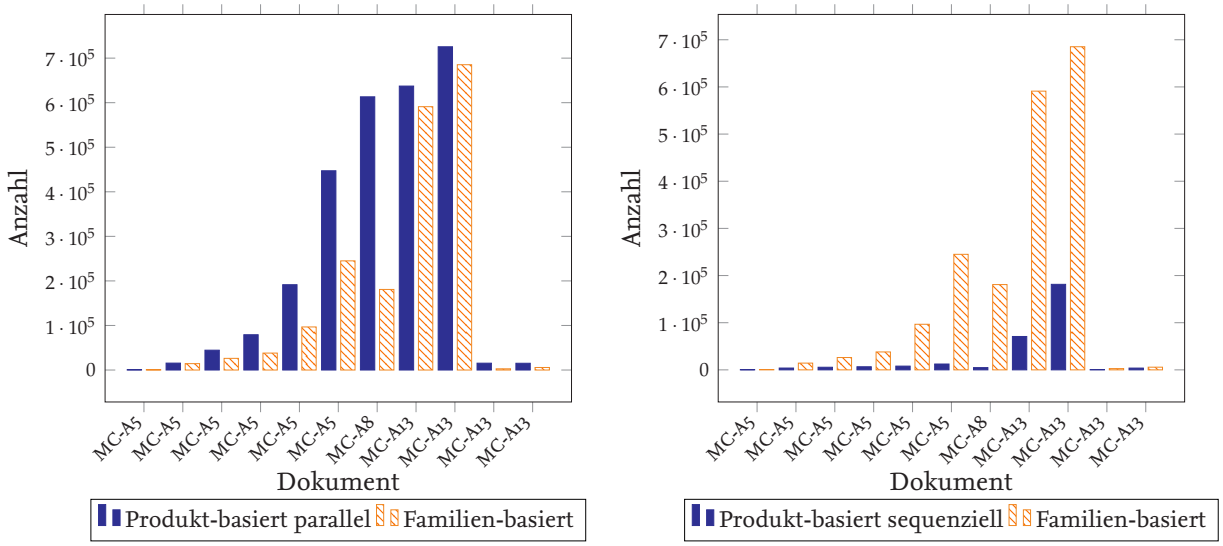


Abbildung 4.5.: Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien in Abhängigkeit der Codegröße

Die Diagramme in Abbildung 4.6a und 4.6b repräsentieren die beim Model Checking explorierten Zustände der Familien-basierten Strategie und der parallelen beziehungsweise sequenziellen Produkt-basierten Strategie für die Hypothesen GH6 beziehungsweise GH7.



(a) Speicherverbrauch FAM & PROD_{PARA}

(b) Speicherverbrauch FAM & PROD_{SEQ}

Abbildung 4.6.: Vergleich der explorierten Zustände beim (a) parallelen Produkt-basierten Model Checking und (b) sequenziellen Produkt-basierten Model Checking mit Familien-basierten Model Checking für die einzelnen Primärstudien

4.2.7. Interpretation der Diagramme und Ergebnisse des Signifikanztests

In diesem Abschnitt werden die Meta-Diagramme aus Abschnitt 4.2.6 sowie die Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests aus Tabelle 4.21 interpretiert. Die Interpretationen sind in Tabelle 4.22 aufgelistet.

Die Tabelle 4.22 enthält in der ersten Spalte die Bezeichnung der untersuchten Hypothese sowie eine Referenz auf das zugehörige Meta-Diagramm. In der zweiten Spalte der Tabelle sind Erkenntnisse und Erläuterungen zum Trend des jeweiligen Meta-Diagramms dargestellt. Beispielsweise ist in der ersten Zeile in der zweiten Spalte der Trend erläutert, dass die Familien-basierte Strategie größtenteils schneller ist als die Produkt-basierte Strategie. Die dritte Spalte repräsentiert Ausnahmen für den allgemein zu beobachtenden Trend. Dazu sind gegebenenfalls entsprechende Datenpunkte in dem jeweiligen Meta-Diagramm markiert. Exemplarisch ist hierfür ebenfalls die erste Zeile der Tabelle, in dessen Spalte *Ausnahmen* die Markierung *A1* aufgelistet ist, bei dem die Produkt-basierte Strategie schneller ist als die Familien-basierte Strategie. In der Spalte mit der Bezeichnung *Interpretation des Wilcoxon-Tests* ist gekennzeichnet, ob die untersuchte Nullhypothese anhand des Prüfwertes abgelehnt werden kann. Die Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests sind ausführlich in der Tabelle 4.21 aufgelistet, welche auch schon die Entscheidung für oder gegen die Ablehnung der Nullhypothese enthält. Nach welchem Kriterium die Nullhypothese beim Wilcoxon-Test abgelehnt wird, ist in Abschnitt 4.1.1 genau beschrieben. In Kurzform beschrieben wird die Nullhypothese abgelehnt, wenn der berechnete Prüfwert kleiner als der kritische Wert ist. Die Tabelle 4.21 enthält ebenfalls noch den berechneten p-Wert, der bei jedem durchgeführten Hypothesentests die ermittelte Ablehnung beziehungsweise Nicht-Ablehnung der Nullhypothese in allen Fällen bestätigt. Die letzte Spalte der Tabelle 4.22 verdeutlicht nochmal, ob die in Abschnitt 3.4.2 formulierten Hypothesen bezüglich der Verwendung bestimmter Verifikationsstrategien für das Model Checking zutreffen. An dieser Spalte ist erkennbar, dass die Hypothesen GH1, GH3, GH6

Hypothese & Diagramm	Diagramm Trend	Ausnahmen	Interpretation des Wilcoxon-Tests	Hypothese bestätigt?
GH1 Abb. 4.3a	■ FAM schneller als PROD	■ **A1:** PROD schneller als FAM	Nullhypothese abgelehnt	Ja
GH2 Abb. 4.3b	■ Deutlich weniger Datenpunkte ■ 3 von 5 Datenpaaren: FAM schneller als PROD	■ **A1:** PROD geringfügig schneller als FAM ■ **A2:** FAM nur geringfügig schneller als PROD	Nullhypothese nicht abgelehnt	Nein
GH3 Abb. 4.4a	■ FAM schneller als PROD	■ Keine erkennbar	Nullhypothese abgelehnt	Ja
GH4 Abb. 4.4b	■ Deutliche kleinere Datengrundlagen ■ Verifikationszeit-Unterschiede geringer als für viele Features ■ Je mehr Features desto höhere Verifikationszeit	■ **A1:** PROD geringfügig schneller als FAM ■ **A2 & A3:** PROD & FAM nahezu gleich	Nullhypothese nicht abgelehnt	Nein
GH5 Abb. 4.5	■ Kein Trend erkennbar - Verifikationszeit von SPLs mit weniger LoC teilweise größer als von SPLs mit mehr LoC	-	-	-
GH6 Abb. 4.6a	■ FAM weniger Zustände als $PROD_{PARA}$	■ **Studie 1 & 2 (je MC-A 5):** FAM & $PROD_{PARA}$ nahezu gleich	Nullhypothese abgelehnt	Ja
GH7 Abb. 4.6b	■ $PROD_{SEQ}$ deutlich weniger Zustände als FAM	■ **1. Studie (MC-A 5) & letzten 2 Studien (MC-A 13):** FAM & $PROD_{SEQ}$ nahezu gleich groß	Nullhypothese abgelehnt	Ja

Tabelle 4.22.: Interpretation der Untersuchungsergebnisse bezüglich Model Checking

und GH7 für Model Checking nachgewiesen wurden. Bei der Untersuchung der Hypothese GH7 ist anzumerken, dass für die sequenzielle Produkt-basierte Strategie Mittelwerte verwendet wurden. In den einzelnen Primärstudien waren ausschließlich die Summen an Zuständen vom Model Checking aller Produkte enthalten und keine Angaben zu den benötigten Zuständen pro Produkt. Daher wurde die Anzahl der Zustände für die Produkt-basierte Strategie durch die Anzahl der Produkte dividiert, um den Mittelwert zu erhalten. Theoretisch müsste für den Vergleich der sequenziellen Produkt-basierten Strategie mit der Familien-basierten Strategie das Produkt mit der größten Anzahl an Zuständen verwendet werden. Bei der Untersuchung der Hypothese GH5 ist kein Trend für die Abhängigkeit der Performanz einer bestimmten Strategie von der Codegröße erkennbar. Außerdem wurde nur in einer Primärstudie Angaben bezüglich der Menge an Codezeilen dokumentiert, weshalb keine Betrachtung dieser Hypothese über mehrere Ansätze möglich ist. Somit konnte keine Untersuchung der Hypothese GH5 durchgeführt werden. Die Hypothesen GH2 und GH4 konnten mittels des Wilcoxon-Vorzeichen-Rang-Tests bezüglich Model Checking nicht angenommen werden.

4.2.8. Untersuchung der nicht bestätigten Hypothesen

In der Tabelle 4.22 ist dargestellt, dass die Hypothesen GH2 und GH4 nicht bestätigt werden konnten. Für diese beiden Hypothesen wurde dementsprechend jeweils die gegenteilige Hypothese formuliert und untersucht. Diese gegenteiligen Hypothesen sind in Tabelle 4.24 aufgelistet.

Gegenteilige Hypothese	Nullhypothese	Alternativhypothese	Signifikanzniveau
$GH2_{\neg}$	$H0_{GMC2} : FAM \geq PROD$	$H1_{GMC2} : FAM < PROD$	$\alpha = 0,05$
$GH4_{\neg}$	$H0_{GMC4} : FAM \geq PROD$	$H1_{GMC4} : FAM < PROD$	$\alpha = 0,05$

Tabelle 4.24.: Gegenteilige Nullhypothesen und Alternativhypothesen sowie das Signifikanzniveau für die Hypothesen bezüglich Model Checking

Diese Hypothesen wurden anschließend ebenfalls mit dem Wilcoxon-Vorzeichen-Rang-Test untersucht, woraus die Ergebnisse in Tabelle 4.26 resultieren.

Nullhypothese	w	n	$w_{\alpha,n}$	Nullhypothese abgelehnt, wenn $w \leq w_{\alpha,n}$	p-Wert	Ablehnung anhand des p-Werts
$H0_{GMC2}$	6	6	2	$6 \leq 2 \rightarrow$ Nein	0,219	Nein
$H0_{GMC4}$	11	14	25	$11 \leq 25 \rightarrow$ Ja	0,00336	Ja

Tabelle 4.26.: Ergebnisse der Untersuchung der gegenteiligen Hypothesen mit dem Wilcoxon-Vorzeichen-Rang-Test

In Tabelle 4.26 ist dargestellt, dass die Nullhypothese der gegenteiligen Hypothese $GH2_{\neg}$ nicht abgelehnt werden kann. Somit kann mit den vorhandenen Daten weder bestätigt werden, dass die Produkt-basierte Strategie signifikant schneller ist als die Familien-basierte Strategie, noch umgekehrt. Bei der gegenteiligen Hypothese $GH4_{\neg}$ kann die Nullhypothese jedoch anhand des Entscheidungskriteriums des Wilcoxon-Tests abgelehnt werden. Das bedeutet, dass anhand der verfügbaren

Primärdaten die Familien-basierte Strategie signifikant schneller ist als die Produkt-basierte Strategie für Software-Produktlinien mit wenigen Features. Somit konnte das Gegenteil der ursprünglichen Hypothese GH4 bestätigt werden.

4.2.9. Untersuchungsergebnisse bezüglich der Behauptungen der Primärstudien

In diesem Abschnitt werden die Ergebnisse der Hypothesenuntersuchungen mit den Behauptungen und Hypothesen der Primärstudien aus Abschnitt 3.4.1 in Beziehung gesetzt. Dabei werden nur die Behauptungen betrachtet, zu denen Aussagen anhand der Untersuchungsergebnisse formuliert werden können. In den Signifikanztests der Primärstudie MC-A 10 zeigte sich für eine Software-Produktlinie eine signifikant geringere Verifikationszeit des Single-Wise-Samplings gegenüber der Familien-basierten Strategie. Dieses Ergebnis stimmt mit der Behauptung *PH1* überein. Für alle anderen Sampling-Strategien zeigte sich kein signifikanter Vorteil, jedoch weisen die verschiedenen Sampling-Strategien trotzdem teilweise geringere Verifikationszeiten als die Familien-basierte Strategie auf, welche jedoch nicht signifikant sind. Die Behauptung *PH2*, dass die Familien-basierte Strategie außer für Software-Produktlinien mit wenigen Produkten schneller ist als die Produkt-basierte Strategie, zeigt sich anhand der Untersuchungsergebnisse nur teilweise. Insgesamt ist die Familien-basierte Strategie für Model Checking schneller als die Produkt-basierte Strategie, entgegen der Behauptung von Kolesnikov et al. [65] in Bezug auf Software-Produktlinien mit wenigen Produkten zeigt sich für beide Richtungen kein signifikanter Vorteil. Daher lässt sich die Einschränkung der Behauptung *PH2* nicht bestätigen. Jedoch existiert eine geringe Anzahl an Evaluierungsergebnissen, bei denen für wenige Produkte die Produkt-basierte Strategie schneller ist als die Familien-basierte Strategie. In Bezug auf die Behauptung *PH7* bezüglich des Pair-Wise-Samplings ließ sich teilweise eine signifikant geringere Verifikationszeit dieser Sampling-basierten Strategie gegenüber der Familien-basierten Strategie berechnen. Ebenfalls ist das Pair-Wise-Sampling teilweise schneller als die Familien-basierte Strategie und umgekehrt, wobei die Unterschiede dabei nicht signifikant sind. Die *Behauptung 1* aus dem Abschnitt 3.4.1 besagt, dass der Aufwand der Familien-basierten Strategie umso größer ist desto größer die Implementierungsartefakte sind. Diese Behauptung kann anhand des Meta-Diagramms aus Abbildung 4.5 nicht bestätigt werden, da in diesem Diagramm dieser Zusammenhang nicht erkennbar ist. Für die verfügbaren Evaluierungsergebnisse ist der Aufwand der Familien-basierte Strategie mit weniger Codezeilen teilweise sogar größer als der Aufwand für Software-Produktlinien mit mehr Lines of Code.

4.3. Untersuchung der Hypothesen bezüglich Type Checking

Dieser Abschnitt beschreibt die Untersuchungsergebnisse der Hypothesen bezüglich Type Checking. Zu Beginn dieses Abschnitts werden die verwendeten Primärstudien zur Evaluierung der Verifikationsstrategien für das Type Checking zusammengefasst. Anschließend werden die in den einzelnen Primärstudien verglichenen Strategien auf signifikante Unterschiede untersucht. Darauffolgend werden die durchgeführten Untersuchungen der Hypothesen aufgelistet, die Ergebnisse mittels Diagrammen und Tabellen dargestellt und interpretiert. Außerdem werden die Gefahren für die Validität der einzelnen Primärstudien bezüglich Type Checking zusammengefasst und die Gefahren für die Validität der durchgeführten Hypothesentests aufgelistet.

4.3.1. Zusammenfassung der Primärstudien

Insgesamt wurden 18 Primärstudien bezüglich der empirischen Evaluierung der Verifikationsstrategien zum Type Checking von Software-Produktlinien gefunden, bestehend aus den Ergebnissen der Literaturrecherche und den Primärstudien aus der Studie von Thüm et al. [79]. Detaillierte Eigenschaften der einzelnen Primärstudien sind im Anhang A in der Tabelle A.2 dargestellt. In allen 18 dieser Dokumente wurde die Verifikationszeit gemessen, die zum Type Checking benötigt wurde. In keinen dieser Studien wurde die abhängige Variable *Speicherverbrauch* betrachtet und in lediglich einem Ansatz wird zusätzlich der Speed Up angegeben [43]. Deshalb konnten die Hypothesen GH6 und GH7 bezüglich Type Checking nicht untersucht werden. Die Verifikationszeit wurde dabei immer in einer Zeiteinheit wie Sekunden oder Millisekunden angegeben. In drei Studien wurde die Verifikationszeit inklusive einer Setup-Zeit für Präprozessierung oder ähnliche Vorgänge angegeben [65, 24, 61]. Dabei besteht diese Setup-Zeit in der Studie von Kolesnikov et al. [65] sowohl aus der Zeit zum Parsen der Features als auch aus der Zeit zur Komposition der Features. Dahingegen setzt sich die Setup-Zeit in der Arbeit von Claus Hunsen [24] aus allen zusätzlichen Aufgaben zusammen, wie beispielsweise der Initialisierung von Datenstrukturen. Die Kompositionszeit ist in der Arbeit von Claus Hunsen [24] nicht in der Setup-Zeit enthalten, wird jedoch zusätzlich separat evaluiert. Demgegenüber betrachtet Peter Lutz [61] in seiner Arbeit die Zeit zur Konstruktion des abstrakten Syntaxbaums zusätzlich zur Verifikationszeit.

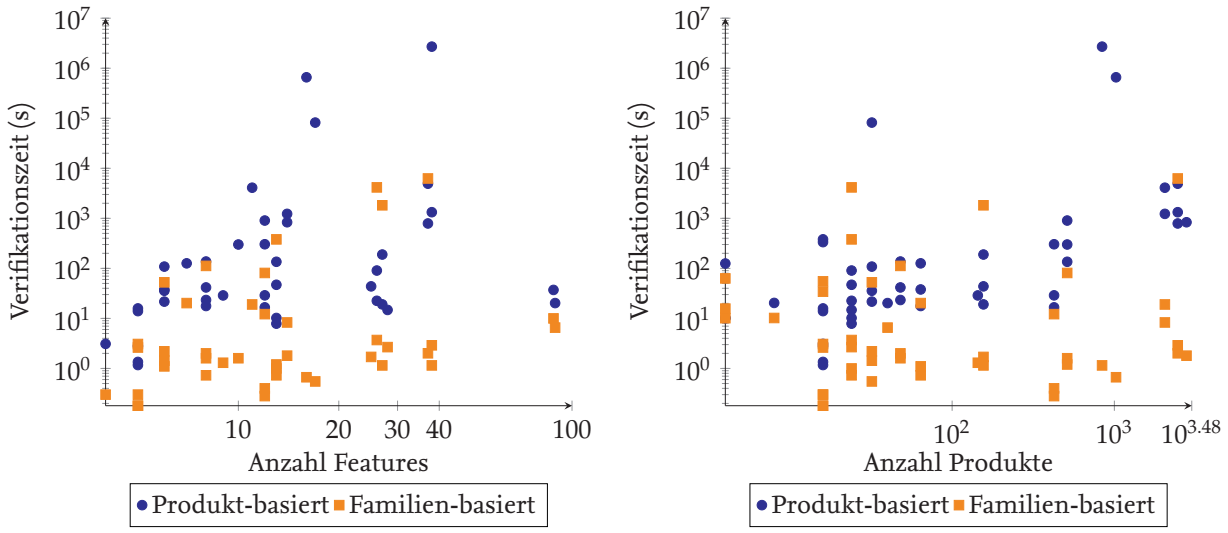
Des Weiteren unterscheiden sich die Primärstudien hinsichtlich der verglichenen Verifikationsstrategien. In sechs Studien wird die Familien-basierte Strategie mit dem Produkt-basierten Ansatz verglichen [66, 52, 11, 20, 67, 69]. Außerdem enthält eine Studie zusätzlich zur Produkt-basierten und Familien-basierten Strategie empirische Evaluierungen für sowohl die Feature-basierte Strategie als auch die Feature-Produkt-basierte Strategie [65]. In einer weiteren Primärstudie werden ebenfalls die Produkt-basierte, die Familien-basierte und die Feature-Produkt-basierte Strategie miteinander verglichen [24]. Darüber hinaus wird in drei Primärstudien die Familien-basierte Strategie mit drei gebräuchlichen Sampling-Strategien miteinander verglichen [43, 42, 41]. Demgegenüber beinhalteten sechs dieser Studien nur eine alleinige Evaluierung der Familien-basierten Strategie [49, 62, 47, 54, 19, 32].

Ein weiterer Unterschied zwischen den Evaluierungen in den Primärstudien ist die Anzahl der zur Evaluierung verwendeten Software-Produktlinien. In zwölf Studien wurden mehrere unterschiedliche Software-Produktlinien verwendet, wobei die Anzahl zwischen drei und zwölf Software-Produktlinien liegt. Diese Software-Produktlinien unterscheiden sich des Weiteren in der Anzahl der Features, der Anzahl der Produkte und der Anzahl an Codezeilen. Alles umfassend enthielten diese Software-Produktlinien vier [65] bis 6918 Features [43, 41, 42], wobei der größte Teil zwischen fünf und 38 Features enthielt. Die Anzahl der Produkte lag dabei insgesamt zwischen zwei [49] und etwa $6,5 * 10^{175}$ [43, 41], der Großteil der Software-Produktlinien beinhaltete 16 bis 512 Produkte. Die Anzahl der Quellcodezeilen unterschied sich dabei ebenfalls stark. Die kleinste Software-Produktlinie bestand aus 11 Codezeilen [52], die größte enthielt 6,7 Millionen Codezeilen [42]. Die meisten Software-Produktlinien der Evaluierungen bestanden aus 2000 bis 260000 Codezeilen.

4.3.2. Allgemeine Meta-Diagramme

Die drei folgenden Diagramme in Abbildung 4.7a, Abbildung 4.7b und Abbildung 4.8 zeigen einen Vergleich der Familien-basierten Strategie und der Produkt-basierten Strategie in Abhängigkeit der

Anzahl Features, der Anzahl Produkte beziehungsweise der Anzahl Codezeilen. In allen drei Diagrammen ist ein allgemeiner Trend erkennbar, in der die Familien-basierte Strategie schneller ist als die Produkt-basierte Strategie.



(a) Zeit vs. Features (b) Zeit vs. Produkte

Abbildung 4.7.: Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit (a) der Anzahl der Features und (b) der Anzahl der Produkte

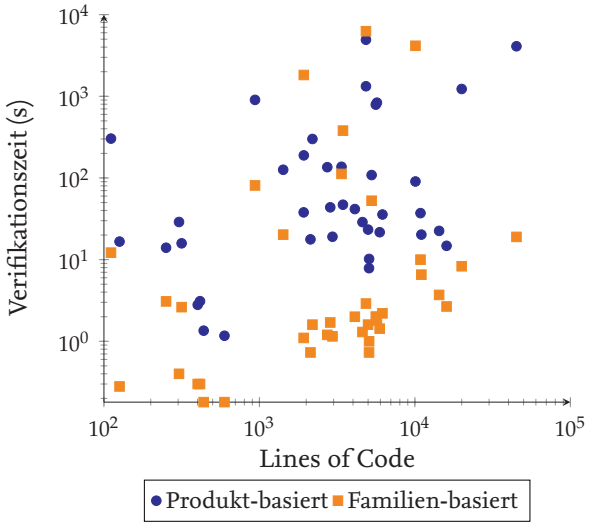


Abbildung 4.8.: Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien in Abhängigkeit der Codegröße

In den drei vorherigen Diagrammen sind einige große Datenteile nicht enthalten, um die Übersichtlichkeit der Abbildungen zu erhalten. Beispielsweise sind Software-Produktlinien mit sehr vielen Produkten wie $6,5 * 10^{175}$ nicht mit visualisiert worden. Die gesamten Datensätze sind im Anhang A in den Tabellen A.26 und A.27 aufgelistet.

4.3.3. Vote Counting der Primärstudien

Die Tabelle 4.28 zeigt die Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die Untersuchungen bezüglich eines signifikanten Vorteils der Familien-basierten Strategie für das Type Checking gegenüber der Produkt-basierten Strategie. Insgesamt konnte nur in fünf Primärstudien der Wilcoxon-Vorzeichen-Rang-Test durchgeführt werden, wobei in all diesen Studien die Stichprobe aus mehreren für die Evaluierung verwendeten Software-Produktlinien bestand. In den restlichen Primärstudien wurde nur eine einzelne Verifikationsstrategie evaluiert oder der Stichprobenumfang war zu gering, weswegen für diese Studien kein Signifikanztest durchgeführt werden konnte. In vier der fünf Primärstudien wurde mit diesem Hypothesentest gezeigt, dass die Familien-basierte Strategie signifikant schneller ist als die Produkt-basierte Strategie. Demgegenüber konnte dieser Zusammenhang in einer Studie nicht bestätigt werden.

Dokument	w	n	$w_{\alpha,n}$	**Nullhypothese abgelehnt, wenn** $w \leq w_{\alpha,n}$	p**-Wert**	**Ablehnung anhand des** p**-Werts**
TC-A 1	0	12	17	$0 \leq 17 \rightarrow$ Ja	0,000244	Ja
TC-A 11	0	5	0	$0 \leq 0 \rightarrow$ Ja	0,0313	Ja
TC-A 13	0	12	17	$0 \leq 17 \rightarrow$ Ja	0,000244	Ja
TC-A 14	0	7	3	$0 \leq 3 \rightarrow$ Ja	0,00781	Ja
TC-A 16	37	11	13	$37 \leq 13 \rightarrow$ Nein	0,65	Nein

Tabelle 4.28.: Ergebnisse des Vote Countings der einzelnen Primärstudien in Bezug auf signifikanter Verbesserung der Verifikationszeit der Familien-basierten Strategie verglichen mit der Produkt-basierten Strategie

Wie bereits erwähnt wurden in den Primärstudien zum Type Checking auch weitere Strategien miteinander verglichen. Auch diese Strategien wurden auf signifikante Unterschiede mit Hilfe des Wilcoxon-Vorzeichen-Rang-Tests überprüft. Die Ergebnisse dieser Überprüfung sind in der Tabelle 4.30 dargestellt. Die Tabelle enthält neben den Parametern und berechneten Werten der Wilcoxon-Vorzeichen-Rang-Tests auch die überprüfte Hypothese. Beispielsweise bedeutet die Hypothese in der ersten Zeile der Tabelle, dass die Feature-basierte Strategie signifikant schneller ist als die Produkt-basierte Strategie. Dieses Schema gilt in der gesamten Tabelle, sodass in jedem Test untersucht wird, ob die Strategie auf der linken Seite des Kleiner-Gleich-Zeichens signifikant schneller ist als die Strategie auf der rechten Seite. Diese Hypothesen wurden intuitiv formuliert, um die Primärstudien auf eine signifikant geringere Verifikationszeit einer bestimmten Strategie gegenüber einer weiteren Strategie zu untersuchen. Dabei wurden diese Hypothesen an den Behauptungen aus Kapitel 3.4.2 sowie an weiteren in während dieser Arbeit gewonnenen Erkenntnisse angelehnt. Wenn eine der Hypothesen bestätigt werden kann, enthält die letzte Spalte das ✓-Zeichen, andernfalls das ✗-Zeichen.

Dokument	w	n	$w_{\alpha,n}$	Überprüfte Hypothese	p-Wert	Hypothese bestätigt?
TC-A 1	0	12	17	FEAT $\leq$ PROD	0,000244	✓
TC-A 1	6,5	12	17	FAM $\leq$ FEAT	0,00602	✓
TC-A 1	21	12	17	FEAT-PROD $\leq$ PROD	0,0881	✗
TC-A 13	78	12	17	FEAT-PROD $\leq$ PROD	1	✗
TC-A 1	0	12	17	FAM $\leq$ FEAT-PROD	0,000244	✓
TC-A 13	0	12	17	FAM $\leq$ FEAT-PROD	0,000244	✓
TC-A 1	0	12	17	FEAT $\leq$ FEAT-PROD	0,000244	✓

Tabelle 4.30.: Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die einzelnen Primärstudien bezüglich der weiteren miteinander verglichenen Strategien

4.3.4. Cohens' d zum Vergleich der Primärstudien

Der Forest Plot in Abbildung 4.9 zeigt die berechnete standardisierte Effektgröße d_{Cohen} für den Vergleich der Verifikationszeit der Familien-basierten Strategie und der Produkt-basierten Strategie. Eine negative Effektgröße symbolisiert einen positiven Effekt der Familien-basierten Strategie verglichen mit der Produkt-basierten Strategie. Ein positiver Effekt bedeutet eine geringere Verifikationszeit. In der Abbildung zeigen fünf von sechs Studien einen positiven Effekt durch die Familien-basierte Strategie. Lediglich eine Studie weist einen negativen Effekt durch die Verwendung der Familien-basierten Strategie auf.

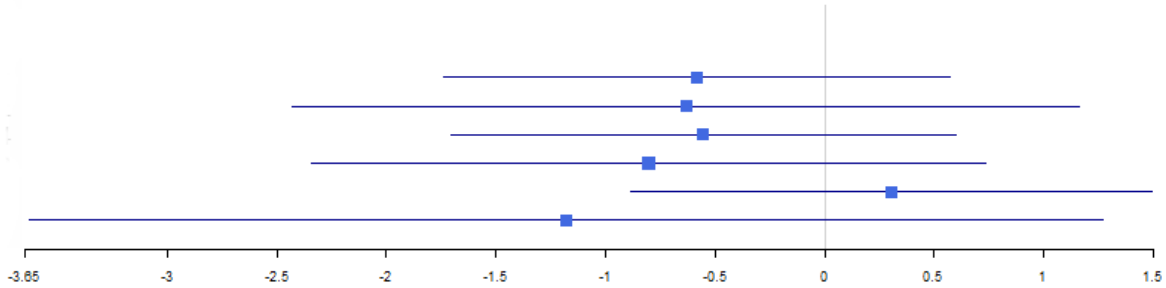


Abbildung 4.9.: Effektgröße Cohen's d der einzelnen Type-Checking-Primärstudien bezüglich der Verifikationszeit

Die Tabelle 4.33 zeigt die Einordnung der ermittelten Effektgrößen in die Interpretationsmetriken. Wie bereits erwähnt weist eine Primärstudie einen kleinen negativen Effekt für die Familien-basierte Strategie auf. Bei drei Primärstudien erzeugt die Familien-basierte Strategie eine mittlere positive Effektgröße. Jeweils eine Studie zeigt einen großen beziehungsweise sehr großen Effekt der Familien-basierten Strategie.

d-Wert Intervall	Bedeutung	Anzahl
$(0,0;2,0]$	negativer Effekt FAM	*1* [61]
$(-0,8;-0,5]$	mittlerer positiver Effekt FAM	*3* [65, 20, 24]
$[-1,0;-0,8]$	großer positiver Effekt FAM	*1* [67]
$(-\infty;-1,0)$	sehr großer positiver Effekt FAM	*1* [69]

Tabelle 4.33.: Zusammenfassung der d_{Cohen}-Werte der einzelnen Type-Checking-Primärstudien

Zusätzlich wurden für die weiteren in den Primärstudien miteinander verglichenen Verifikationsstrategien ebenfalls die Effektstärke d_{Cohen} berechnet. Die Ergebnisse sind im Anhang A in der Tabelle A.20 einsehbar. In dieser Tabelle bedeutet ein negativer numerischer Wert der Effektstärke dass die Verifikationszeit der Strategie B geringer ist als die der Strategie A. Beispielsweise weist die Feature-basierte Strategie in der Primärstudie von Kolesnikov et al. [65] einen Effekt der Größe $-0,577$ gegenüber der Produkt-basierten Strategie auf. Damit zeigt die Feature-basierte Strategie in dieser Studie einen positiven Effekt gegenüber der Produkt-basierten Strategie in Form einer geringeren Verifikationszeit.

4.3.5. Untersuchte Hypothesen

Um die Hypothesen aus Abschnitt 3.4.2 bezüglich Type Checking zu untersuchen, müssen diese Hypothesen in eine für den Wilcoxon-Vorzeichen-Rang-Test passende Form umgewandelt werden. Die entsprechenden Nullhypothesen und Alternativhypothesen inklusive dem verwendeten Signifikanzniveau sind in Tabelle 4.35 aufgelistet.

Hypothese	Nullhypothese	Alternativhypothese	Signifikanzniveau
GH1	$H0_{TC1} : FAM \geq PROD$	$H1_{TC1} : FAM < PROD$	$\alpha = 0,05$
GH2	$H0_{TC2} : PROD \geq FAM$	$H1_{TC2} : PROD < FAM$	$\alpha = 0,05$
GH3	$H0_{TC3} : FAM \geq PROD$	$H1_{TC3} : FAM < PROD$	$\alpha = 0,05$
GH4	$H0_{TC4} : PROD \geq FAM$	$H1_{TC4} : PROD < FAM$	$\alpha = 0,05$
GH5	$H0_{TC5} : FAM \geq PROD$	$H1_{TC5} : FAM < PROD$	$\alpha = 0,05$

Tabelle 4.35.: Nullhypothesen, Alternativhypothesen und das Signifikanzniveau für die Hypothesen bezüglich Type Checking

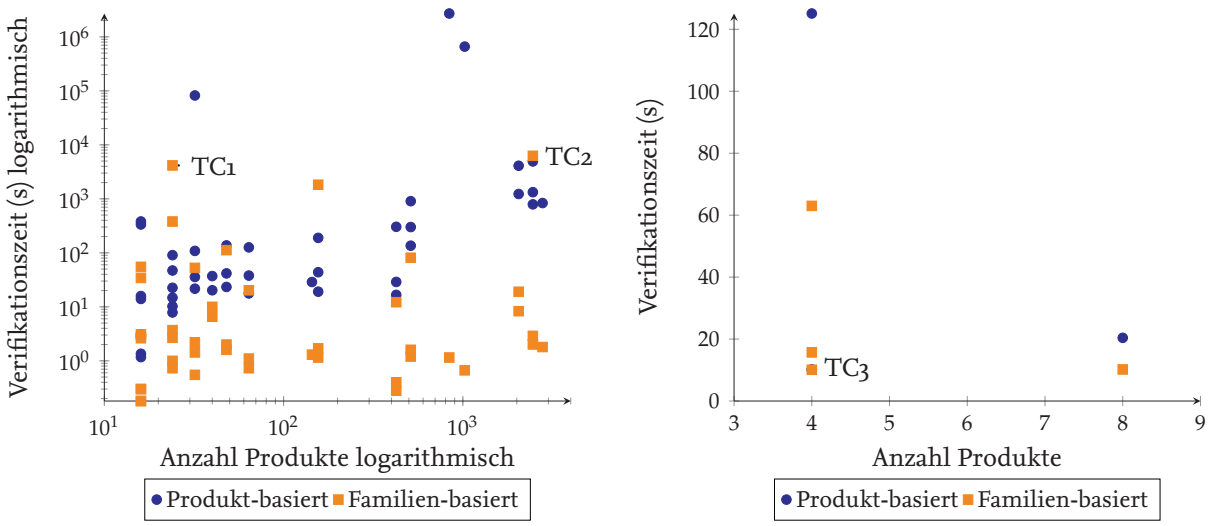
Nachdem die Nullhypothesen und Alternativhypothesen formuliert wurden, wurde der Wilcoxon-Vorzeichen-Rang-Test durchgeführt. Die Parameter sowie Ergebnisse des Tests sind in der Tabelle 4.37 aufgeführt. Die Spalten enthalten von links nach rechts das Kürzel der überprüften Hypothese, den durch den Test berechneten Prüfwert w, die Stichprobengröße n, sowie den kritischen Wert $w_{\alpha,n}$, die Ablehnungsentscheidung für die Nullhypothese als auch den p-Wert und die Interpretation des p-Werts.

Null-hypothese	w	n	$w_{\alpha,n}$	Nullhypothese abgelehnt, wenn $w \leq w_{\alpha,n}$	p-Wert	Ablehnung anhand des p-Werts
$H0_{TC1}$	149	46	389	$149 \leq 389 \rightarrow$ Ja	$2{,}69 * 10^{-6}$	Ja
$H0_{TC2}$	-	4	-	-	-	-
$H0_{TC3}$	101	34	200	$101 \leq 200 \rightarrow$ Ja	0,000232	Ja
$H0_{TC4}$	55	10	10	$55 \leq 10 \rightarrow$ Nein	1	Nein
$H0_{TC5}$	68	31	163	$68 \leq 163 \rightarrow$ Ja	$9{,}77 * 10^{-}8$	Ja

Tabelle 4.37.: Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die Hypothesen bezüglich Type Checking

4.3.6. Meta-Diagramme der Hypothesen

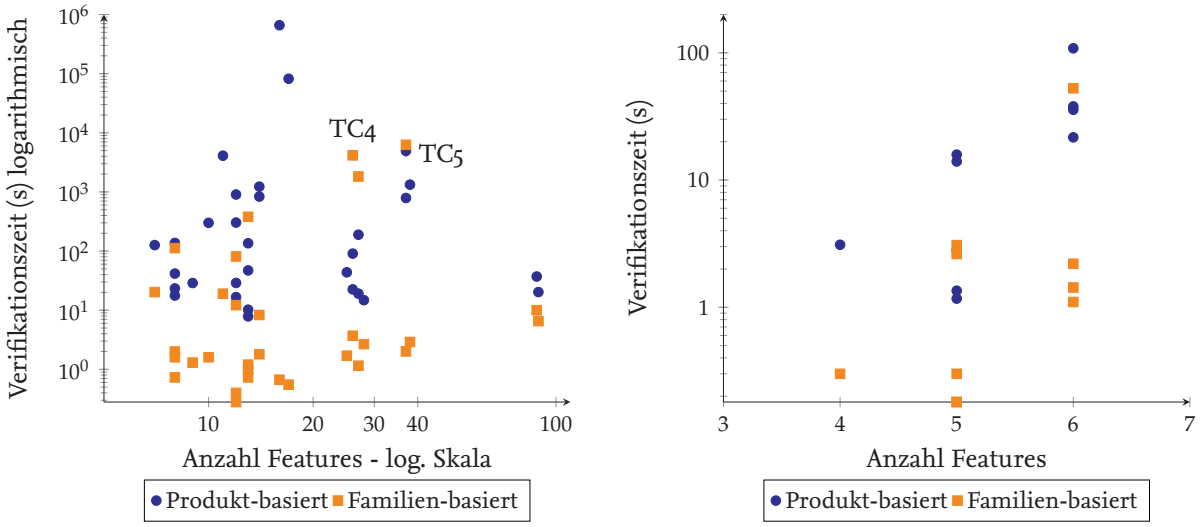
In diesem Unterkapitel werden in den Abbildungen 4.10a bis 4.12 der Meta-Diagramme für die Verifikationszeit des Type Checkings in Bezug auf die Kategorien von wenigen und vielen unabhängigen Variablen dargestellt. Das Meta-Diagramm in Abbildung 4.10a visualisiert die Verifikationszeit der Produkt-basierten und Familien-basierten Strategie für Software-Produktlinien mit vielen Produkten, um das Verhältnis zwischen diesen beiden Strategien für viele Produkte in Bezug auf die Hypothese GH1 darzustellen. Dabei ist zu beachten, dass sehr große abhängige und unabhängige Variablen der Übersichtlichkeit halber nicht in den Diagrammen enthalten sind. Die gesamten Datensätze sind in den Tabellen A.21 bis A.25 im Anhang A aufgeführt.



(a) Zeit FAM & PROD (b) Zeit FAM & PROD

Abbildung 4.10.: Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten für Software-Produktlinien mit (a) vielen Produkten und (b) wenigen Produkten

Die Abbildungen 4.11a und 4.11b stellen die Meta-Diagramme für die Hypothesen GH3 und GH4 für die Verifikationszeit für viele beziehungsweise wenige Features.



(a) Zeit FAM & PROD (b) Zeit FAM & PROD

Abbildung 4.11.: Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien mit (a) vielen Features und (b) wenigen Features

Die Abbildung 4.12 zeigt die Evaluierungsergebnisse der Verifikationszeit der Produkt-basierten Strategie und Familien-basierten Strategie in Bezug für viele Codezeilen zur Untersuchung der Hypothese GH5.

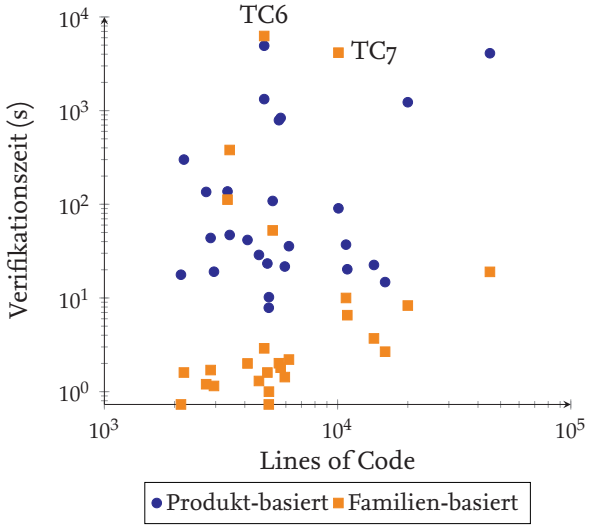


Abbildung 4.12.: Vergleich der Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie für Software-Produktlinien mit mehr als 2000 Codezeilen

4.3.7. Interpretation der Diagramme und Ergebnisse des Signifikanztests

Die Meta-Diagramme aus Abschnitt 4.3.6 und die Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests aus Tabelle 4.37 werden in diesem Kapitel interpretiert. Die Interpretationsergebnisse sind in tabellarischer Form in Tabelle 4.38 dargestellt.

Die erste Spalte der Tabelle 4.38 enthält das Kürzel der untersuchten Hypothese und die Abbil-

Hypothese & Diagramm	Diagramm Trend	Ausnahmen	Interpretation des Wilcoxon-Tests	Hypothese bestätigt?
GH1 Abb. 4.10a	■ FAM schneller als PROD	■ **TC1:** FAM langsamer ■ **TC2:** PROD geringfügig schneller als FAM	Nullhypothese abgelehnt	✓
GH2 Abb. 4.10b	■ Zu wenig Datenpunkte für einen Signifikanztest ■ Größtenteils FAM schneller als PROD	■ **TC3:** PROD & FAM etwa gleich schnell	Kein Test möglich	-
GH3 Abb. 4.11a	■ FAM schneller als PROD	■ **TC4:** FAM langsamer ■ **TC5:** PROD & FAM etwa gleich schnell	Nullhypothese abgelehnt	✓
GH4 Abb. 4.11b	■ Deutliche kleinere Stichprobe ■ Je mehr Features desto höhere Verifikationszeit ■ FAM immer schneller als PROD	■ Keine	Nullhypothese nicht abgelehnt	✗
GH5 Abb. 4.12	■ FAM größtenteils schneller als PROD	■ **TC6:** FAM langsamer als PROD ■ **TC7:** FAM langsamer als PROD	Nullhypothese abgelehnt	✓

Tabelle 4.38.: Interpretation der Untersuchungsergebnisse bezüglich Type Checking

dung des zugehörigen Meta-Diagramms. Die nächste Spalte listet Erkenntnisse basierend auf den Meta-Diagrammen auf. In der ersten Zeile ist beispielsweise die Erkenntnis aufgeführt, dass ein Trend in diesem Meta-Diagramm erkennbar ist, in dem die Familien-basierte Strategie größtenteils schneller ist als die Produkt-basierte Strategie. Die dritte Spalte enthält mögliche Ausnahmen die dem allgemeinen Trend gegenüber stehen. Dazu sind gegebenenfalls entsprechende Datenpunkte in dem jeweiligen Meta-Diagramm markiert. Exemplarisch ist hierfür ebenfalls die erste Zeile der Tabelle, in der die Markierung *TC1* aufgelistet ist, bei dem die Familien-basierte Strategie langsamer ist als die Produkt-basierte Strategie. Die Spalte mit dem Titel *Interpretation des Wilcoxon-Tests* zeigt, ob die untersuchte Nullhypothese anhand des Prüfwertes abgelehnt werden kann. Die gesamten Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests sind in der Tabelle 4.37 aufgelistet. Das Ablehnungskriterium einer Nullhypothese beim Wilcoxon-Test ist in Abschnitt 4.1.1 genau beschrieben. Zusammengefasst erfolgt die Ablehnung der Nullhypothese, wenn der berechnete Prüfwert kleiner als der kritische Wert ist. In der letzten Spalte der Tabelle 4.22 wird mittels der Zeichen ✓ und ✗ deutlich dargestellt, ob die formulierten Hypothesen bezüglich der Verwendung bestimmter Verifikationsstrategien für das Type Checking bestätigt werden können. In der zweiten Zeile ist ersichtlich, dass zur Hypothese GH2 keine Wilcoxon-Vorzeichen-Rang-Test möglich war, da die Stichprobe eine Größe von vier hat und somit nicht verwendet werden konnte (siehe Abschnitt 4.1.1). Anhand der letzten Spalte ist erkennbar, dass die Hypothesen GH1, GH3 und GH5 bezüglich Type Checking bestätigt werden konnten. Die Hypothese GH4 konnte mittels des Wilcoxon-Vorzeichen-Rang-Tests nicht nachgewiesen werden, da der berechnete Prüfwert größer als der kritische Wert war.

4.3.8. Untersuchung der nicht bestätigten Hypothesen

Wie am Ende des Abschnitts 4.3.7 zur Interpretation der Untersuchungsergebnisse der Hypothesen formuliert, konnte die Hypothese GH4 nicht bestätigt werden. Aus diesem Grund wurde für diese Hypothese die gegenteilige Hypothese formuliert und untersucht. Diese gegenteilige Hypothese ist in Tabelle 4.40 aufgelistet.

Gegenteilige Hypothese	Nullhypothese	Alternativhypothese	Signifikanzniveau
$Gh4_{\neg}$	$H0_{GTC4} : FAM \geq PROD$	$H1_{GTC4} : FAM < PROD$	$\alpha = 0,05$

Tabelle 4.40.: Gegenteilige Nullhypothesen und Alternativhypothesen sowie das Signifikanzniveau für die Hypothesen bezüglich Model Checking

Diese Hypothesen wurden anschließend ebenfalls mit dem Wilcoxon-Vorzeichen-Rang-Test untersucht, woraus die Ergebnisse in Tabelle 4.26 resultieren.

Null-hypothese	w	n	$w_{\alpha,n}$	**Nullhypothese abgelehnt, wenn** $w \leq w_{\alpha,n}$	p**-Wert**	**Ablehnung anhand des** p**-Werts**
$H0_{GTC4}$	11	14	25	$11 \leq 25 \rightarrow$ Ja	0.003357	Ja

Tabelle 4.42.: Ergebnisse der Untersuchung der gegenteiligen Type-Checking-Hypothese mit dem Wilcoxon-Vorzeichen-Rang-Test

In Tabelle 4.42 dargestellt, dass bei der gegenteiligen Hypothese $GH4_{\neg}$ die Nullhypothese anhand des Entscheidungskriteriums des Wilcoxon-Tests abgelehnt werden. Das bedeutet, dass anhand der verfügbaren Primärdaten die Familien-basierte Strategie signifikant schneller ist als die Produkt-basierte Strategie für Software-Produktlinien mit wenigen Features. Somit konnte das Gegenteil der ursprünglichen Hypothese GH4 bestätigt werden.

4.3.9. Untersuchungsergebnisse bezüglich der Behauptungen der Primärstudien

In diesem Abschnitt werden die Ergebnisse der Hypothesenuntersuchungen mit den Behauptungen und Hypothesen der Primärstudien verglichen. Dabei wird nur der Bezug der Untersuchungsergebnisse auf die Hypothesen und Behauptungen aus Abschnitt 3.4.1 erläutert, zu denen Aussagen anhand der Untersuchungsergebnisse formuliert werden können. Die Behauptung *PH1*, dass Sampling-basierte Strategien schneller sein können als die Familien-basierte Strategie, konnte mittels des Signifikanztests nicht untersucht werden. Jedoch besitzen einige Sampling-Strategien, wie beispielsweise das Single-Conf-Sampling, teilweise eine geringere Verifikationszeit in den Evaluierungen der Primärstudien als die Familien-basierte Strategie. Allerdings sind gegenteilige Messergebnisse ebenfalls vorhanden, in denen die Familien-basierte Strategie schneller ist. Außerdem zeigte sich in den Untersuchungen die *Behauptung 1* aus Abschnitt 3.4.1. Das Single-Conf-Sampling weist eine geringere Verifikationszeit in den Evaluierungen auf und die Familien-basierte Strategie ist in den einzelnen Primärstudien schneller als das Pair-Wise-Sampling. Diese Evaluierungsergebnisse ließen sich allerdings aufgrund zu kleiner Stichproben nicht mit dem Signifikanztest untersuchen. Insgesamt zeigte sich die Familien-basierte Strategie signifikant schneller als die Produkt-basierte Strategie für Type Checking. Dieses Ergebnis stimmt mit der Hypothese *PH2* überein. Für die Teilhypothese bezüglich Software-Produktlinien mit wenigen Produkten konnte kein Signifikanztest durchgeführt werden. Bei der Betrachtung der Verifikationszeiten beider Strategien für wenige Produkte zeigte sich teilweise die Familien-basierte Strategie und in einem Fall die Produkt-basierte Strategie als schneller. Entgegen der Behauptung *PH3* von Kolesnikov et al. [65] ließ sich für die Primärstudie TC-A 1 von Kolesnikov et al. [65] das Gegenteil zeigen. Unter Berücksichtigung der Setup-Zeit ist die Familien-basierte Strategie in der Primärstudie TC-A 1 signifikant schneller als die Feature-basierte Strategie. Ohne Betrachtung der Setup-Zeit ist die Feature-basierte Strategie teilweise geringfügig schneller und teilweise ist die Familien-basierte Strategie schneller. Die *Behauptung 1* lässt sich anhand Abbildung 4.8 interpretieren. In dem Meta-Diagramm in Abbildung 4.8 nimmt die Verifikationszeit mit dem Anstieg an Codezeilen zu und stimmt somit mit *Behauptung 1* von Brabrand et al. [23] überein.

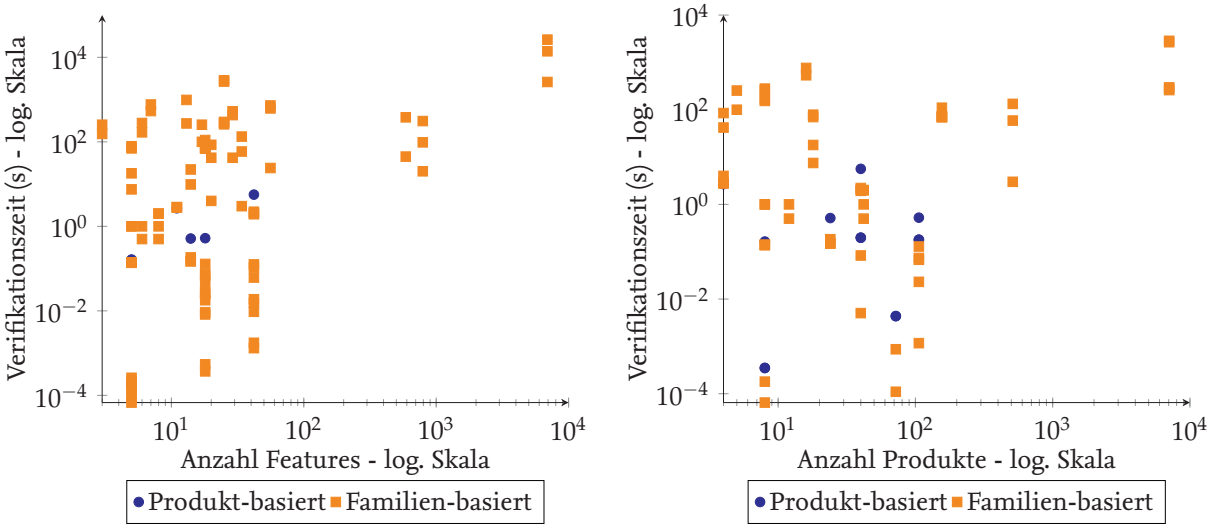
4.4. Untersuchung der Hypothesen bezüglich Static Analysis

Dieser Abschnitt enthält die Untersuchungen bezüglich der Verifikationsstrategien für Software-Produktlinien unter Verwendung von Static Analysis. Am Anfang dieses Kapitels erfolgt eine kurze Zusammenfassung der für diese Arbeit relevanten Daten der Primärstudien bezüglich Static Analysis. Des Weiteren erfolgt eine Untersuchung der einzelnen Primärstudien, in denen Evaluierungsergebnisse verschiedener Verifikationsstrategien enthalten sind, auf signifikante Vorteile einer dieser Strategien gegenüber einer anderen Verifikationsstrategie. Da für den Bereich der statischen Analyse keine ausreichenden Evaluierungsergebnisse zur Überprüfung möglicher Hypothesen vorlagen, wird die Abhängigkeit der vorhandenen Verifikationsstrategien für Static Analysis ausschließlich mittels Streudiagrammen dargestellt und interpretiert.

In Abschnitt 3.3.3 wurde beschrieben, dass insgesamt elf Primärstudien für den Bereich der statischen Analyse ermittelt wurden. Die Evaluierungen in allen elf Primärstudien untersuchten alle die Verifikationszeit einer oder mehrerer Verifikationsstrategien und ausschließlich in Zeiteinheiten zwischen Mikrosekunden und Minuten. In einer Studie wurde dabei zusätzlich die Generierungszeit der einzelnen Produkte beziehungsweise des Metaprodukts betrachtet [23]. Der Speicherverbrauch wurde während der Evaluierung in zwei Primärstudien untersucht [23, 31]. Außerdem wurde der Speed Up einer Strategie gegenüber einer weiteren Strategie ebenfalls in zwei Primärstudien erläutert [34, 43].
Ein weiterer Unterschied zwischen diesen 11 Primärstudien besteht in den betrachteten Verifikationsstrategien. Ein Vergleich der Familien-basierten Strategie und der Produkt-basierten Strategie wurde in zwei Primärstudien durchgeführt [23, 34]. In sechs Primärstudien wurde einzig die Familien-basierte Strategie evaluiert [27, 37, 16, 17, 31, 3]. Darüber hinaus wurden in drei Primärstudien die Familien-basierte Strategie und drei gebräuchliche Sample-basierte Strategien gegenübergestellt [41, 42, 43].
In allen elf Primärstudien wurden jeweils mehrere Software-Produktlinien für die empirische Evaluierung verwendet. Diese Software-Produktlinien unterscheiden sich in der Anzahl der Features, der Anzahl der Produkte und der Anzahl an Codezeilen. Die kleinste Menge an Features einer dieser Software-Produktlinien betrug drei Features [31], die größte Software-Produktlinie enthielt 6918 Features [41, 42, 43]. Die Mehrzahl der zur Evaluierung verwendeten Software-Produktlinien enthielt zwischen fünf und 42 Features. Insgesamt lag die Anzahl der Produkte dabei zwischen vier [23, 27] und etwa $6,5 * 10^{175}$ [41, 43] und der Großteil der Software-Produktlinien beinhaltete acht bis 512 Produkte. Dahingegen bestand die kleinste Software-Produktlinie aus 257 Codezeilen [37], die größte enthielt 6,7 Millionen Codezeilen [42]. Die meisten der in den Primärstudien verwendeten Software-Produktlinien besaßen 1000 bis 84000 Codezeilen. Die detaillierte Eigenschaften und relevanten Daten der einzelnen Primärstudien sind im Anhang A in der Tabelle A.3 aufgelistet.

In den Abbildungen 4.13a, 4.13b und 4.14 ist die Verifikationszeit aller empirischen Evaluierungen der einzelnen Verifikationsstrategien der Primärstudien bezüglich der statischen Analyse dargestellt. Die Messdaten dieser Diagramme sind in Anhang A in den Tabellen A.26 und A.27 einzusehen. In diesen drei Meta-Diagrammen sind einige sehr große Werte der jeweiligen abhängigen und unabhängigen Variablen entfernt worden, um die Übersichtlichkeit der Diagramme zu erhal-

ten. Die entfernten Daten sind in den beiden entsprechenden Tabellen der Messdaten markiert.
Das Streudiagramm in Abbildung 4.13a zeigt die Verifikationszeit in Abhängigkeit von der Anzahl der Features. Die Verifikationszeit der Produkt-basierten Strategie wurde nur in zwei Primärstudien evaluiert, daher ist der Anteil dieser Messwerte in dem Diagramm sehr gering und erlaubt keine Behauptung eines allgemeinen Performanztrends der Produkt-basierten Strategie. Für die Familien-basierte Strategie lässt sich ein Anstieg der Verifikationszeit durch Erhöhung der Anzahl an Features erkennen, wobei sogar die Behauptung einer potenziellen linearen Abhängigkeit der Verifikationszeit von der Anzahl der Features möglich wäre.
Das Diagramm in Abbildung 4.13b enthält ebenfalls deutlich weniger empirische Daten bezüglich der Verifikationszeit der Produkt-basierten Strategie verglichen mit den Daten der Familien-basierten Strategie. Auch für das Verhältnis der Produkt-basierten Verifikationszeit und der Anzahl der Produkte ist durch die geringe Datengrundlage keine Interpretation möglich. Die Performanz in Form der Verifikationszeit nimmt bei der Familien-basierten Strategie mit zunehmender Anzahl Produkte im Allgemeinen zu. Außerdem ist erkennbar, dass die Familien-basierte Strategie schneller ist als die Produkt-basierte Strategie für die selbe Anzahl an Produkten.
Die Abbildung 4.14 zeigt ebenfalls keinen Trend des Verhältnisses der Verifikationszeit der Produkt-basierten Strategie und der Größe des Quellcodes aufgrund der kleinen Datenmenge. Für die Familien-basierte Strategie ist ebenfalls keine Interpretation möglich. In der Abbildung 4.14 ist erkennbar, dass die Verifikationszeit für größere Software-Produktlinien kleiner ist als die Verifikationszeit für kleinere Software-Produktlinien.



(a) Zeit vs. Features (b) Zeit vs. Produkte

Abbildung 4.13.: Darstellung der gesamten verfügbaren Daten zur Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit (a) der Anzahl der Features und (b) der Anzahl der Produkte

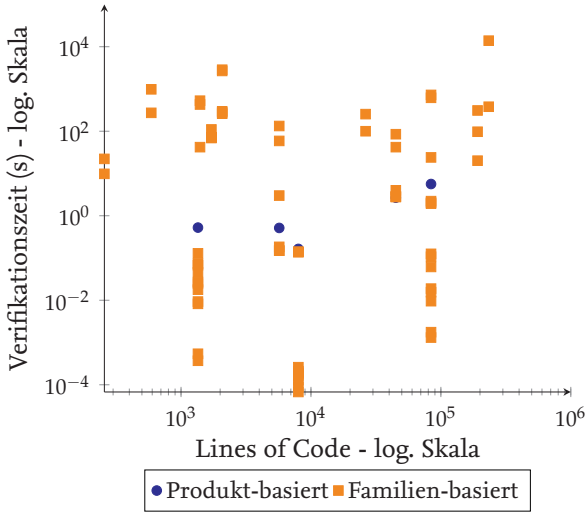


Abbildung 4.14.: Darstellung der gesamten verfügbaren Daten zur Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit der Codegröße

Des Weiteren wurde für die Primärstudie von Brabrand et al. [23] jeweils der Unterschied zwischen den drei vorgeschlagenen Familien-basierten Ansätzen mit der Verifikationszeit des Produkt-basierten Ansatzes auf Signifikanz untersucht. Diese Primärstudie bietet als einzige der elf Static-Analysis-Primärstudien die Möglichkeit für einen Signifikanztest mittels des Wilcoxon-Vorzeichen-Rang-Tests. Die Ergebnisse sind in der Tabelle 4.44 dargestellt. Die Tabelle zeigt das Ergebnis des Signifikanztests der Verifikationszeit jeweils einer der drei Familien-basierten Strategien gegenüber der Verifikationszeit der Produkt-basierten Strategie. Für alle drei Familien-basierte Ansätze kann keine signifikant geringere Verifikationszeit gegenüber des Produkt-basierten Ansatzes gezeigt werden, weder anhand des Ablehnungskriterium des Wilcoxon-Vorzeichen-Rang-Tests noch anhand des p-Werts.

Dokument	w	n	$w_{\alpha,n}$	**Nullhypothese abgelehnt, wenn** $w \leq w_{\alpha,n}$	p**-Wert**	**Ablehnung anhand des** p**-Werts**
SA-A 1	2	5	0	$2 \leq 0 \rightarrow$ Nein	0,0938	Nein
SA-A 1	2	5	0	$2 \leq 0 \rightarrow$ Nein	0,0938	Nein
SA-A 1	2	5	0	$2 \leq 0 \rightarrow$ Nein	0,0938	Nein

Tabelle 4.44.: Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests der Primärstudie SA-A 1 bezüglich einer signifikant geringeren Verifikationszeit der Familien-basierten Strategie verglichen mit der Produkt-basierten Strategie

4.5. Untersuchung der Hypothesen bezüglich Theorem Proving

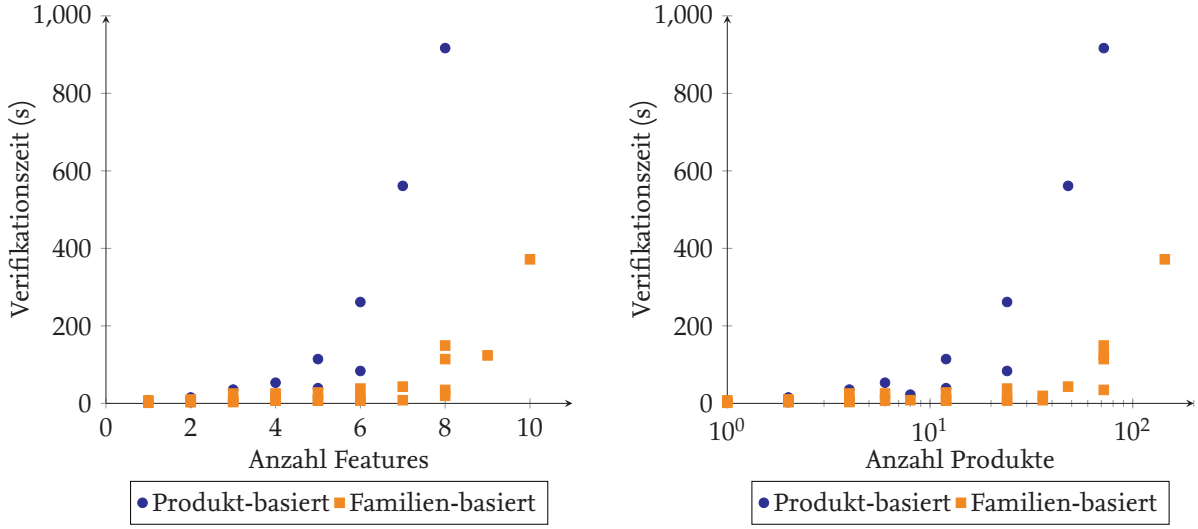
In diesem Kapitel wird das Verhältnis der Ansätze zum Theorem Proving unter Verwendung verschiedener Verifikationsstrategien untersucht. Als erstes werden die für diese Untersuchung relevanten Daten der einzelnen Primärstudien zusammengefasst. Des Weiteren werden für Primärstu-

dien, in denen mehrere Verifikationsstrategien evaluiert wurden, Signifikanztests durchgeführt und die Ergebnisse erläutert. Aufgrund der geringen Menge an Primärstudien für den Bereich Theorem Proving konnten keine Hypothesenuntersuchungen durchgeführt werden, sodass lediglich der Trend der Verifikationszeit der verschiedenen Verifikationsstrategien mittels Diagrammen visualisiert und anschließend erläutert wird.

Insgesamt wurden sechs Primärstudien für die Evaluierung der Verifikationsstrategien bezüglich Theorem Proving für Software-Produktlinien ermittelt, welche in Tabelle A.4 aufgelistet sind. In jeder dieser Primärstudien wurde die Verifikationszeit evaluiert. In einer Studie wurde die Verifikationszeit nicht in einer üblichen Einheit wie beispielsweise Sekunden gemessen, sondern in der Anzahl der *Lines of Proof*, der Menge an notwendigen Zeilen zur Durchführung eines Beweises [76]. In zwei dieser Studien wurde zusätzlich zur Verifikationszeit die Generierungszeit des Metaprodukts für die Familien-basierte Strategie ermittelt [53, 36]. In keiner der sechs Studien wurden andere Effektstärken wie Speicherverbrauch Speed Up berichtet.
In den Primärstudien werden unterschiedliche Ansätze basierend auf unterschiedlichen Verifikationsstrategien vorgestellt und evaluiert. Die Familien-basierte Strategie wird in zwei Primärstudien mit der Produkt-basierten Strategie verglichen [77, 36], in einer Studie wird ein Ansatz zur Feature-Produkt-basierten Strategie vorgestellt und mit der Produkt-basierten Strategie gegenübergestellt [76]. Drei Ansätze verfolgen die Familien-basierte Strategie und evaluieren ausschließlich den Familien-basierten Ansatz [13, 53, 78].
Darüber hinaus besteht ein Unterschied zwischen den Primärstudien in der Anzahl der für die Evaluierung verwendeten Software-Produktlinien. In allen sechs wissenschaftlichen Dokumenten wird lediglich eine Software-Produktlinie zur Evaluierung genutzt. Dabei ist in drei dieser Studien diese Software-Produktlinie in verschiedenen Variationen in Form verschiedener Anzahl Features und Produkte genutzt wurden [77, 36, 78]. Die Anzahl der Features der Software-Produktlinien lag zwischen zwei [77, 36, 78] und zehn Features [78]. Die Menge an Produkten der Software-Produktlinien betrug zwischen zwei [77, 36, 78] und 144 Elementen [78], wohingegen die Mehrzahl der Software-Produktlinien zwischen zwei und 72 Produkten enthielt. Alle zur Evaluierung verwendeten Software-Produktlinien bestanden aus 51 [36] bis 12.209 Codezeilen [13]. Der Großteil dieser Software-Produktlinien besaß zwischen 75 und 239 Lines of Code. Die detaillierten Eigenschaften und relevanten Daten der einzelnen Primärstudien sind im Anhang A in der Tabelle A.4 aufgelistet.

Die Abbildungen 4.15a, 4.15b und 4.16 stellen die Verifikationszeit der Produkt-basierten und der Familien-basierten Strategie der Primärstudien in Relation zu der Anzahl der Features, der Anzahl der Produkte beziehungsweise der Anzahl an Lines of Code dar. Die Rohdaten, auf denen diese Diagramme basieren, sind im Anhang A in den Tabellen A.28 und A.29 aufgelistet. In jeder dieser Abbildungen steigt sowohl die Verifikationszeit der Produkt-basierten Strategie als auch der Familien-basierten Strategie mit der steigenden Anzahl an Produkten an. Dabei ist in Abbildung 4.15a die Verifikationszeit dieser beider Strategien in etwa gleich groß, bei steigender Anzahl an Features ist die Familien-basierte Strategie deutlich schneller als die Produkt-basierte Strategie. Derselbe Trend zeigt sich in Abbildung 4.15b für die Performanz der beiden Strategien in Verbindung mit der Anzahl an Produkten sowie in der Abbildung 4.16 für die Verifikationszeit dieser Strategien in Abhängigkeit der Codegröße. In allen drei Abbildungen lässt sich der Trend einer Abhängigkeit

der Verifikationszeit der Produkt-basierten beziehungsweise Familien-basierten Strategie und der Anzahl an Features, Anzahl an Produkten beziehungsweise der Quellcodegröße vermuten.



(a) Zeit vs. Features (b) Zeit vs. Produkte

Abbildung 4.15.: Darstellung der gesamten verfügbaren Daten zur Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit (a) der Anzahl der Features und (b) der Anzahl der Produkte

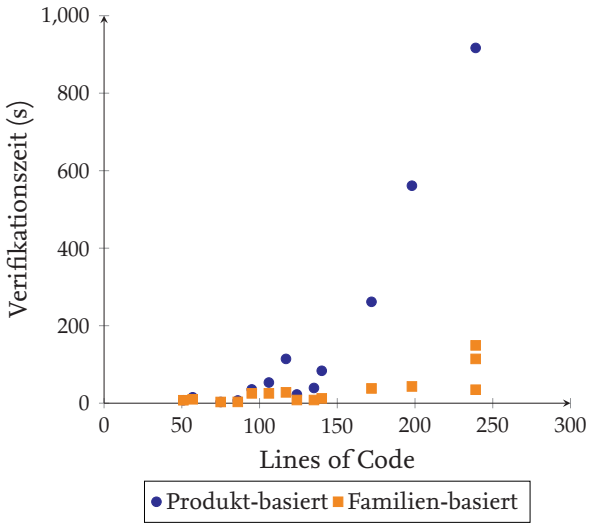


Abbildung 4.16.: Darstellung der gesamten verfügbaren Daten zur Verifikationszeit der Produkt-basierten Strategie und der Familien-basierten Strategie in Abhängigkeit der Codegröße

Außerdem wurde mit dem Wilcoxon-Vorzeichen-Rang-Test untersucht, ob in den Primärstudien von Thüm et al. [77] und von Jens Meinicke [36] die Familien-basierte Strategie signifikant schneller ist als die Produkt-basierte Strategie. Darüber hinaus wurde die Primärstudie Thüm et al. [76] auf eine signifikante Verringerung der Verifikationszeit der Feature-Produkt-basierten Strategie gegenüber der Produkt-basierten Strategie überprüft. Die Ergebnisse sind in Tabelle 4.46 darge-

stellt. Diese Tabelle zeigt, dass die Familien-basierte Strategie in den Primärstudien von Thüm et al. [77] sowie von Jens Meinicke [36] signifikant schneller ist. Allerdings ergab der Test bei der Primärstudie von Thüm et al. [77] einen gerundeten p-Wert von $0,05017$, welcher somit größer als das Signifikanzniveau α mit einem Wert von $0,05$ ist. Daher würde der p-Wert nicht auf die Ablehnung der Nullhypothese hinweisen. Aufgrund des gerundeten p-Werts der gleichen Größe des Prüfwerts und des kritischen Werts wird die Hypothese angenommen, dass die Familien-basierte signifikant schneller ist als die Produkt-basierte Strategie. Bei den anderen beiden Signifikanztests stimmten das Ablehnungskriterium des Signifikanztest sowie die Ablehnung anhand des p-Wertes überein.

Dokument	w	n	$w_{\alpha,n}$	**Überprüfte Hypothese**	p**-Wert**	**Hypothese bestätigt?**
TP-A 1	0	5	0	FAM $\leq$ PROD	~0,05017	✓
TP-A 2	0	6	2	FEAT-PROD $\leq$ PROD	0,01563	✓
TP-A 5	0	8	5	FAM $\leq$ PROD	~0,0113	✓

Tabelle 4.46.: Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die einzelnen Theorem-Proving-Primärstudien

4.6. Gefahren für die Validität

Dieser Abschnitt listet als erstes die Gefahren für die Validität der einzelnen Primärstudien zusammenfassend auf. Dabei werden nur die Gefahren der Validität der Primärstudien bezüglich Model Checking und Type Checking berücksichtigt, da für Static Analysis und Theorem Proving keine Signifikanztests möglich waren. Anschließend werden ebenfalls eventuelle kritische Aspekte der in dieser Arbeit durchgeführten Untersuchungen der formulierten Hypothesen auf erläutert.

4.6.1. Gefahren für die Validität der Model-Checking-Primärstudien

In sieben der 28 Primärstudien wurden Gefahren für die Validität der durchgeführten Experimente und Fallstudien dokumentiert [55, 7, 8, 71, 57, 28, 68]. Nur bei zwei dieser Studien sind die Gefahren der Validität in interne und externe Validität unterteilt [8, 71], eine dieser Studien unterscheidet zwischen interner und externer Validität sowie Konstruktvalidität. Einige der Primärstudien, die Gefahren für die Validität enthalten, erläutern dass für die Untersuchung nur eine Software-Produktlinie oder eine kleine Fallstudie durchgeführt wurde [55, 8, 57]. Die Autoren stellen dabei in Frage, in wie weit das Ergebnis der Untersuchung generalisiert werden kann. Des Weiteren wird oftmals erläutert, dass die unterschiedlichen verwendeten Model Checker und die Toolchain Einflüsse auf die Untersuchungsergebnisse haben können [7, 57, 68]. Besonders die Werkzeuge zum Model Checking besitzen zahlreiche Optimierungsparameter, wobei oftmals nur die grundlegenden Konfigurationsparameter verwendet werden, um den Einfluss durch den Model Checker zu minimieren. Weitere Gefahren für die Validität beziehen sich auf die verwendeten Programmiersprachen [55], auf die Möglichkeit zur Parallelisierung [55], die Anzahl der Experimentdurchläufe um zufällige Einflüsse auszuschließen [55, 8] oder dass die bei der Produkt-basierten Strategie nicht mit enthalten ist [7, 8]. Ausführliche Darstellungen der Gefährdungen der Validität sind in den entsprechenden Primärstudien zu finden.

4.6.2. Gefahren für die Validität der Type-Checking-Primärstudien

Lediglich fünf der 18 Primärstudien zur Evaluierung der Verifikationsstrategien zum Type Checking von Software-Produktlinien enthalten Gefahren für die Validität der durchgeführten Experimente und Fallstudien [65, 43, 24, 41, 42]. In jeder dieser Studien sind die Gefahren der Validität in interne und externe Validität unterteilt. Alle diese Studien formulieren eine Beschränkung auf die Unterstützung einer Teilmenge der Typregeln und Konstrukte der verwendeten Implementierungssprache als Gefahr für die interne Validität. Drei Studien beachten nur die ISO/IEC[2] Typregeln für die Programmiersprache C, jedoch nicht die Erweiterungen durch GNU[3] C [43, 41, 42]. Kolesnikov et al. erläutern das der entwickelte Type Checker nur einen Teil der Typregeln der Programmiersprache Java überprüft [65]. In den Primärstudien, welche die Familien-basiert Strategie mit drei Sampling-basierten Strategien vergleichen, werden teilweise Quelldateien aufgrund technischer Probleme ausgeschlossen und somit nicht die komplette Software-Produktlinie geprüft. Jede dieser Studien nennt die Auswahl und die Menge der zur Evaluierung verwendeten Software-Produktlinien als Gefährdung der externen Validität. Ausführliche Darstellungen der Gefährdungen der Validität sind in den entsprechenden Primärstudien zu finden.

4.6.3. Gefahren für die Validität der Hypothesenuntersuchungen

- In einigen Primärstudien wurden die unabhängigen Variablen wie Anzahl der Features und Anzahl der Produkte nur textuell und ungenau beschrieben [29, 31, 38, 64, 50]. Diese textuellen Beschreibungen wurden dazu verwendet, die Software-Produktlinien mit *FeatureIDE*[4] zu rekonstruieren und damit die Anzahl der Features und Anzahl der Produkte zu bestimmen. Diese Ermittlung der unabhängigen Variablen stellt eine weitere Gefährdung für die interne Validität der Untersuchung der Hypothesen dar. Da diese Beschreibungen teilweise ungenau waren, besteht die Möglichkeit einer fehlerhaften Rekonstruktion des Feature Modells und somit die Gefahr für die Verwendung fehlerhafter Daten. Bei welchen Primärstudien die Anzahl der Features und Produkte rekonstruiert wurden, ist in der Tabelle A.1 dargestellt. Da diese Rekonstruktion jedoch nur bei wenigen Primärstudien nötig war und aufgrund der generell geringen Anzahl an Primärstudien, wurden diese Daten dennoch verwendet.

- Für die Untersuchung der Hypothesen wurden Software-Produktlinien in wenige und viele Features, Produkte und Lines of Code unterteilt. Diese Kategorisierung erfolgte zum einen anhand Betrachtung der Studien, in denen die Produkt-basierte Strategie schneller ist als die Familien-basierte Strategie. Zum anderen wurden die Studien betrachtet, in denen die berechnete Effektgröße d_{Cohen} keinen positiven Effekt der Familien-basierten Strategie aufzeigt. Diese Klassifizierung ist kritisch zu betrachten und gefährdet die interne als auch die externe Validität. Theoretisch wäre eine Bestimmung von *Break-Even-Points* denkbar, allerdings ist es fraglich in wie weit diese Bestimmung für die aggregierten Daten möglich ist. Andererseits könnten die Break-Even-Points der einzelnen Primärstudien verwendet werden. Diese wurden jedoch weder in einer der Primärstudien angegeben noch konnten die Break-Even-Points anhand vorliegender Experimentergebnisse ansatzübergreifend berechnet wer-

[2] http://www.iso.org/iso/standards_development/processes_and_procedures/iso_iec_directives_and_iso_supplement.htm
[3] https://gcc.gnu.org/
[4] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

den. Außerdem ist zum Anfertigungszeitpunkt im Allgemeinen unklar, welche Menge an Features, Produkten oder Codezeilen jeweils wenige beziehungsweise viele dieser unabhängigen Variablen darstellen. Somit war die intuitive Methode die einzige mögliche Variante um die Kategorien der Software-Produktlinien zu bilden, sodass dadurch die Untersuchung überhaupt ermöglicht wurde.

- Bei der Untersuchung der Hypothesen und dem Vergleich der einzelnen Verifikationsstrategien wurde ausschließlich die Performanz in Form von Verifikationszeit oder Speicherverbrauch betrachtet. Eine Berücksichtigung der Validität der Verifikationsergebnisse hat nicht stattgefunden. Dadurch wird die interne Validität der Untersuchung gefährdet. Beispielsweise wurden im Dokument von Cordy et al. [55] die Ergebnisse der Verifikation der einzelnen Eigenschaften für die Familien-basierte und Produkt-basierte Strategie dokumentiert. Theoretisch sollten die Ergebnisse der Produkt-basierten Strategie und der Familien-basierten Strategie keine Unterschiede aufweisen. Bei einem Vergleich mit der Feature-basierten Strategie oder optimierten Produkt-basierten Strategie in Form von Sampling könnten die Ergebnisse in Form von Wahrheitswerten relevant sein, um den wirklichen Nutzen dieser Strategien zu bestimmen. Die Feature-basierte Strategie ermöglicht, wie bereits in dieser Arbeit beschrieben, die Überprüfung von Feature-lokalen Eigenschaften. Und die Sample-basierte Strategie überprüft nur einen Teil der möglichen Produkte und ist somit unvollständig. Somit könnten diese Strategien eventuell schneller als der Produkt-basierte Ansatz oder der Familien-basierte Ansatz sein, dahingegen jedoch nicht alle Fehler entdecken, die diese beiden Strategien aufdecken würden. Da die Produkt-basierte sowie die Familien-basierte Strategie alle möglichen Fehler entdecken, die mittels Model Checking überprüft werden können, wurden die Verifikationsergebnisse im Vergleich dieser Strategien nicht betrachtet. Außerdem enthalten nur wenige der recherchierten Dokumente Angaben zu den Ergebnissen, welche der überprüften Eigenschaften *TRUE* bzw. *FALSE* sind.

- Bei der Berechnung der Effektstärke d_{Cohen} werden Annahmen zur Normalverteilung der Daten und zur Homogenität der Varianzen getätigt. Die empirischen Daten der Primärstudien waren weder normalverteilt noch waren die Varianzen für die miteinander verglichenen Stichproben der verglichenen Strategien gleich groß. Des Weiteren weisen einige dieser empirischen Daten große Standardabweichungen auf. Die großen Standardabweichungen resultieren daraus, dass in den Primärstudien die Verifikationsstrategien an unterschiedlichen Software-Produktlinien evaluiert wurden und sich somit die ermittelten Verifikationszeiten von Produktlinie zu Produktlinie stark unterscheiden können. Die Nichteinhaltung der genannten Voraussetzungen stellt die interne Validität der Berechnung dieser standardisierten Effektgröße in Frage. Ungeachtet der nicht erfüllten Annahmen reflektieren die berechneten Effektgrößen die empirisch ermittelten Unterschiede zwischen den einzelnen verglichenen Strategien in den Primärstudien. Des Weiteren existieren weitere standardisierte Effektgrößen, die jedoch jeweils ähnliche Annahmen und Voraussetzungen aufweisen. Eine Einführung zu diesen weiteren Effektgrößen ist in dem Buch von Stephen Rustenbach [70] enthalten. Da diese Effektgrößen ebenfalls Eigenschaften, wie die Normalverteilung der Daten, voraussetzen und die Effektgröße d_{Cohen} häufig verwendet wird, wurde sich in dieser Arbeit für die Effektgröße d nach Cohen entschieden.

- In der Untersuchung der Hypothesen wurden keine Untergruppen der Ansätze betrachtet, wodurch die interne Validität der Hypothesenuntersuchung in Frage gestellt werden könnte. Theoretisch wäre beispielsweise im Bereich des Model Checkings ein Vergleich der Performanz der verglichenen Strategien möglich, in denen *symbolisches Model Checking* verwendet wird [7, 68]. Die Verfügbarkeit an geeigneten Primärstudien mit passenden Evaluierungsergebnissen für diese Untergruppen ist zu gering für eine solche Meta-Analyse der Untergruppen. Beim Type Checking wären Untergruppen für die verschiedenen Implementierungstechniken für Software-Produktlinien vorstellbar. Beispielsweise fließen sowohl Feature-orientierte Software-Produktlinien mittels AHEAD[5] [62] als auch Software-Produktlinien mit dem CPP Präprozessor [11, 19] in einen gemeinsamen Signifikanztest ein. Da der Fokus dieser Arbeit jedoch auf einer Untersuchung der Vorteile einer bestimmten Verifikationsstrategie für Software-Produktlinien mit bestimmten Eigenschaften liegt, erfolgte keine Unterteilung in diese Untergruppen mit dem Ziel der Erzeugung eines allgemein gültigen Ergebnisses. Außerdem erlaubt die Verfügbarkeit an verwendbaren Primärstudien keine Untersuchung von Untergruppen der anfänglich genannten Unterschiede, da die Anzahl an Primärstudien zur Evaluierung von Verifikationsstrategien für Software-Produktlinien generell sehr klein ist und somit keine Signifikanztests aufgrund zu kleiner Stichproben möglich wären. Somit war ausschließlich eine gemeinsame Betrachtung dieser Studien möglich. In welchem Ausmaß die gemeinsame Betrachtung möglicher Unterkategorien valide ist, stellt eine weitere Forschungsfrage dar, die den Umfang dieser Arbeit überschreitet.

- In den Untersuchungen der Hypothesen hat sich generell ein signifikanter Vorteil der Familien-basierten Strategie gegenüber der Produkt-basierten Strategie gezeigt. Ausgenommen dabei sind die Hypothesen *GH2* und *GH7*. Dennoch erfordert die Familien-basierte Strategie meistens die Erzeugung eines Metaprodukts. Die Erzeugung dieses Metaprodukts erweist sich in der Praxis oftmals als schwierig und ist nicht trivial. Diese Tatsache sollte in der Praxis bei der Bewertung der in diesem Abschnitt gefunden Ergebnisse berücksichtigt werden. Die Ergebnisse sollten das Ziel verfolgen, eine Entscheidung für eine spezifische Strategie zu ermöglichen. Theoretisch zeigt die Familien-basierte Strategie generelle Vorteile gegenüber der Produkt-basierten Strategie. Jedoch wird dieser Vorteil in der Praxis dadurch abgeschwächt, dass die Erzeugung des Metaprodukts schwierig ist. Folglich gefährdet die Nicht-Berücksichtigung der Generierungszeit sowohl aller Produkte als auch des Metaprodukts die interne Validität der Untersuchungen dieser Arbeit.

- Eine Gefährdung der externen Validität der durchgeführten Hypothesentests stellt die Auswahl der verwendeten Primärstudien dar. Da der Rechercheprozess systematisch durchgeführt wurde und zusätzlich zu den häufig ermittelten Konferenzpapieren auch Masterarbeiten [24, 53, 59, 61, 69] und Doktorarbeiten [6, 41] berücksichtigt wurden, sind annähernd alle für diese Arbeit relevanten Studien ermittelt worden. Dadurch bilden die verwendeten Primärstudien eine valide Repräsentation der Allgemeinheit.

- Die aggregierten Datensätze für Software-Produktlinien mit wenigen Features und wenigen Produkten sind relativ klein. Anhand der kleinen Stichprobengröße dieser beiden Kategorien

[5] https://www.cs.utexas.edu/users/schwartz/ATS.html

wird die externe Validität der Hypothesentest-Ergebnisse in Frage gestellt. Da in dieser Arbeit keine eigenen Evaluierungen durchgeführt wurden und der systematische Literaturrecherche-Prozess keine weiteren passenden Primärstudien lieferte, konnten nur diese kleinen Datenmengen für Software-Produktlinien mit wenigen Features und wenigen Produkten verwendet werden.

- Die externe Validität der Signifikanztests und der damit verbundenen Hypothesenuntersuchungen wird dadurch gefährdet, dass nur Teilmengen der empirischen Evaluierungen genutzt werden konnten. Beispielsweise wurden in einigen Primärstudien die Ansätze einer bestimmten Verifikationsstrategie ausschließlich einzeln evaluiert anstatt mit einer anderen Strategie verglichen zu werden. Diese Daten konnten für die Untersuchung der einzelnen Hypothesen nicht verwendet werden, da dazu abhängige Stichproben notwendig sind. Aus dem Grund, dass nur eine Teilmenge dieser Daten verwendet werden konnte, kann die Allgemeingültigkeit der Annahme beziehungsweise Ablehnung einer Hypothese abgeschwächt werden. Allerdings konnten auch durch explizite Nachfrage teilweise keine fehlenden Evaluierungsergebnisse der Primärstudien gewonnen werden, sodass die ausschließliche Möglichkeit zur Untersuchung der Hypothesen die Verwendung der vorliegenden Daten war.

- Während der Untersuchungen wurde kein Einteilung der Art der Software-Produktlinien durchgeführt. Somit wurden sowohl künstlich generierte, akademische als auch industrielle Software-Produktlinien gemeinsam betrachtet. Diese gemeinsame Betrachtung kann die externe Validität der Hypothesenuntersuchungen gefährden. Dadurch wird eine gemeinsame Aussage für diese Arten von Software-Produktlinien formuliert, wobei vor allem eine separate Überprüfung für industrielle Software-Produktlinien interessant wäre. Diese realen, industriellen Software-Produktlinien sind häufig deutlich größer als akademische Software-Produktlinien, welche häufig nur für die durchgeführten Evaluierungen verwendet werden. Da in den Primärstudien größtenteils keine Angaben zur Art der Software-Produktlinien existieren, kann diese Eigenschaft nicht berücksichtigt werden und somit werden alle verwendeten Software-Produktlinien gemeinsam betrachtet.

- In den einzelnen Primärstudien bezüglich Model Checking wurden die Ansätze für unterschiedlich viele Eigenschaften evaluiert [40, 55, 7, 8, 71, 30, 75]. Bei der Betrachtung der einzelnen Evaluierungsergebnisse zeigte sich, dass die Performanz vom Model Checking starke Unterschiede je nach Struktur und Größe der jeweiligen geprüften logischen Formel aufweist. Ein simples Beispiel ist, dass das Model Checking einer längeren und ineinander verschachtelten logischen Formel allgemein deutlich zeitaufwändiger ist. Um diese Abhängigkeit bei Primärstudien, die unterschiedliche logische Formeln für die Evaluierung verwendet haben, zu minimieren, wurde der arithmetische Mittelwert für diese Primärstudien gebildet. Diese Mittelwerte können die interne Validität der Signifikanztests beeinflussen. Durch Verwendung des arithmetischen Mittels werden somit keine Unterschiede je nach geprüfter Formel betrachtet. Dadurch wird nicht untersucht, ob eventuelle Verbesserungen für spezifische logische Formeln durch Verwendung einer bestimmten Strategie gegenüber einer anderen Strategie existieren. Dennoch ermöglicht die Verwendung des arithmetischen Mittels ein Vergleich dieser Primärstudien, in denen nur eine logische Formel zum Model Checking verwendet wurde.

- Einige der Primärstudien bezüglich Type Checking enthielten keine vollständigen Angaben zur Verifikationszeit der Produkt-basierten Strategie [11, 20]. Stattdessen wurde in diesen Primärstudien nur Verifikationszeit pro Produkt für die Produkt-basierte Strategie angegeben. Um diese empirischen Daten für die Hypothesentests verwenden zu können, wurde die Verifikationszeit pro Produkt mit der Anzahl der Produkte multipliziert, um die gesamte Verifikationszeit der Produkt-basierten Strategie in diesen Primärstudien zu erhalten. Üblicherweise ist die Verifikationszeit von Produkt zu Produkt unterschiedlich, wodurch diese ermittelte gesamte Verifikationszeit einen Schätzwert darstellt. Stattdessen würde die Angabe eines Durchschnittswerts der Verifikationszeit aller Produkte eine genauere Berechnung der gesamten Produkt-basierten Verifikationszeit ermöglichen. Ohne eine genaue gesamte Verifikationszeit der Produkt-basierten Strategie könnten diese Teile den Signifikanztest in beiden Richtungen beeinflussen und daher die interne Validität des Hypothesentests gefährden. Zum einen sollte der Schätzwert jedoch einen plausiblen Wert darstellen, da sich die angegeben Zeit zur Verifikation eines Produkts für ein anderes Produkt theoretisch sowohl erhöhen als auch verringern könnte. Zum anderen ergab der Wilcoxon-Vorzeichen-Rang-Test ohne die Berücksichtigung dieser Primärstudien dasselbe Ergebnis.

- In drei Primärstudien zum Type Checking [65, 24, 61] wurde die ermittelte Verifikationszeit inklusive einer separaten Setup-Zeit aufgelistet, welche je nach Studie unterschiedliche zusätzliche Zeiten enthält (siehe Abschnitt 4.3.1). In den restlichen verwendeten Primärstudien wurde ausschließlich die Zeit die zum Type Checking aufgebracht werden musste betrachtet. Diese unterschiedlichen Arten der Betrachtung der Verifikationszeit könnten eine Gefahr für die interne Validität der Hypothesenuntersuchung darstellen. Da der Wilcoxon-Vorzeichen-Rang-Test allerdings Vergleiche und Rangdifferenzen paarweise bildet, wird die interne Validität dadurch weniger stark gefährdet. Denn dadurch werden jeweils somit pro Datenpaar solche mit oder ohne diese Setup-Zeit verglichen. Allerdings bleibt die Frage offen, ob diese verschiedenen Verifikationszeiten gemeinsam betrachtet werden sollten oder nicht. Generell jedoch dürfte die zusätzliche Betrachtung dieser Zeit die Validität nicht gefährden. In der Studie von Kolesnikov et al. [65] ist die Setup-Zeit bei der Produkt-basierten Strategie deutlich größer als die Setup-Zeit der Familien-basierten Strategie. Daher würde die gemeinsame Betrachtung der Verifikationszeit und der Setup-Zeit in der Studie von Kolesnikov et al. [65] dahingehend beeinflussen, dass die Familien-basierte Strategie noch deutlicher eine geringere Verifikationszeit aufweist als die Produkt-basierte Strategie. In der Studie von Claus Hunsen [24] ist die Setup-Zeit pro Evaluierung in etwa so groß wie die Verifikationszeit, wodurch das allgemeine Verhältnis zwischen den verglichenen Strategien nicht verändert werden würde. In der Arbeit von Peter Lutz [61] ist die Setup-Zeit der Produkt-basierten Strategie deutlich größer als die Setup-Zeit der Familien-basierten Strategie. Diese gemeinsame Betrachtung hätte denselben Effekt wie der zuvor erläuterte Einfluss der Verwendung dieser gemeinsamen Zeiten in der Arbeit von Kolesnikov et al. [65]. Daher sollte die gemeinsame Betrachtung der Setup-Zeit und Verifikationszeit keine negativen Auswirkungen auf die Ergebnisse der Hypothesenuntersuchungen haben.

- Die Ansätze für die verschiedenen Verifikationsstrategien zum Type Checking von Software-Produktlinien können teilweise nicht alle Typregeln der verwendeten Programmiersprache

prüfen [43, 41, 42, 65]. Für den Signifikanztest wurde diese Unvollständigkeit nicht berücksichtigt, da die verwendeten Primärstudien einerseits teilweise keine Angaben dazu enthielten, welche Typregeln überprüft werden können. Dies könnte die interne Validität bedrohen. Üblicherweise können mit bestehenden Type Checkern für einzelne Produkte die kompletten Typregeln überprüft werden und somit könnte die Produkt-basierte Strategie eventuell einen Vorteil bezüglich der Vollständigkeit aufweisen. Dadurch könnte sich bei einer Prüfung der vollständigen Typregeln das Ergebnis eventuell zu Gunsten der Produkt-basierten Strategie verschieben.

4.7. Zusammenfassung der Untersuchungsergebnisse

In den ersten vier Abschnitten dieses Kapitels wurden die Hypothesen für spezifische Verifikationstechniken untersucht. Diese Ergebnisse werden in der folgenden Tabelle 4.48 nochmals zusammengefasst und danach zur Bildung einer allgemeinen Aussage bezüglich der Verifikationsstrategien interpretiert.

Globale Hypothese	Model Checking	Type Checking	Static Analysis (Trend)	Theorem Proving (Trend)
GH1	✓	✓	(✓)	(✓)
GH2	✗	-	∼	∼
GH3	✓	✓	∼	(✓)
GH4	✗	✗	∼	∼
GH5	-	✓	∼	(✓)
GH6	✓	-	-	-
GH7	✓	-	-	-

Tabelle 4.48.: Ergebnisse Hypothesenuntersuchungen der einzelnen Verifikationsmethoden

In der Tabelle 4.48 sind in der ersten Spalte die globalen Hypothesen aufgeführt, die in Abschnitt 3.4.2 formuliert wurden. In den weiteren Spalten der Tabelle ist aufgelistet, ob diese jeweiligen Hypothesen für die einzelnen Verifikationsmethoden bestätigt oder abgelehnt wurden. Dabei ist anzumerken, dass für die Verifikationsmethoden Static Analysis und Theorem Proving keine Hypothesen untersucht wurden. Die Aussagen dieser beiden Analysemethoden basieren ausschließlich auf den ermittelten Trends in den entsprechenden Meta-Diagrammen. In der Tabelle 4.48 haben die Symbole ✓ und ✗ die bisherige Bedeutung. Wenn der Trend der Meta-Diagramme bezüglich Theorem Proving oder Static Analysis auf die Bestätigung einer Hypothese hindeutet, ist das ✓ von Klammern umschlossen, da diese Ergebnisse wie bereits erwähnt jeweils nur anhand der Diagramme vermutet werden. Ein horizontaler Strich in einer Tabellenzelle bedeutet, dass zu dieser Hypothese für die entsprechende Analyse aufgrund unzureichender Datengrundlage keine Untersuchung durchgeführt werden konnte. Wenn anhand der Meta-Diagramme der Bereiche Static Analysis und Theorem Proving keine Aussage bezüglich einer Hypothese möglich ist, wird dies durch das Symbol ∼ dargestellt.

Die Hypothese GH1 ist mit hoher Wahrscheinlichkeit bestätigt, da sie in allen Analysebereichen per Signifikanztest oder Diagramminterpretation bestätigt beziehungsweise vermutet wurde. Dass die

Produkt-basierte Strategie bei wenigen Produkten signifikant schneller als die Familien-basierte Strategie ist, kann dahingegen nicht bestätigt werden. Diese Hypothese GH2 wurde für keine Analysetechnik bestätigt. Demgegenüber wurde für Model Checking und Type Checking ein signifikanter Vorteil der Familien-basierten Strategie im Vergleich zur Produkt-basierten Strategie bezüglich der Verifikationszeit von Software-Produktlinien mit vielen Features bestätigt. Zusätzlich deuten die Meta-Diagramme für Static Analysis and Theorem Provin ebenfalls die Bestätigung dieser Hypothese an. Somit kann die Hypothese GH3 höchstwahrscheinlich angenommen werden. Die Hypothese GH4 wurde für keine Analysetechnik bestätigt, sodass für Software-Produktlinien mit wenigen Features die Produkt-basierte Strategie nicht signifikant schneller sein sollte als die Familien-basierte Strategie. Entgegen dieser Hypothese konnte sogar für Model Checking und Type Checking das Gegenteil berechnet werden, dass die Familien-basierte Strategie für wenige Features ebenfalls signifikant schneller ist als die Produkt-basierte Strategie. Dass die Familien-basierte Strategie für Software-Produktlinien mit vielen Lines of Code signifikant schneller ist, zeigt ein gemischtes Ergebnis in den einzelnen Bereichen. Im Bereich Model Checking enthielten wenige Evaluierungen Angaben zur Anzahl Codezeilen, sodass keine Untersuchung möglich war. Für die Static Analysis war aufgrund einer geringen Datengrundlage keine Aussage möglich. Für Type Checking zeigt sich dahingegen ein signifikanter Vorteil der Familien-basierten Strategie und das Meta-Diagramm für Theorem Proving veranschaulicht, dass die Familien-basierte Strategie schneller ist als die Produkt-basierte Strategie.

In Bezug auf den Speicherverbrauch waren nur im Bereich Model Checking ausreichende Evaluierungen vorhanden, in den restlichen Bereichen wurde diese abhängige Variable gar nicht oder nur sehr gering betrachtet. Daher ist fraglich, in wie weit sich das Ergebnis der Hypothesen zum sequenziellen und parallelen Speicherverbrauch auf eine allgemeine Aussage erweitern lässt. Dennoch lässt das Ergebnis zu den Hypothesen GH6 und GH7 die Vermutung zu, dass das Ergebnis vom Model Checking auf die anderen Teilbereiche und die Allgemeinheit in Form der Hypothesen GH6 und GH7 übertragbar ist.

Durch die Ergebnisse dieser Hypothesenuntersuchung und der in diesem Unterkapitel erläuterten Möglichkeit zur Verallgemeinerung der Teilergebnisse kann eine Antwort für die Forschungsfrage F1 hergeleitet werden. Anhand der verfügbaren empirischen Evaluierungsergebnisse der einzelnen Primärstudien ist somit die Familien-basierte Strategie besser bezüglich der Verifikationszeit als die Produkt-basierte Strategie für alle betrachteten Eigenschaften von Software-Produktlinien. Lediglich für Software-Produktlinien mit wenigen Produkten zeigte sich für keine dieser beiden Strategien ein signifikanter Vorteil. Daher sind für Software-Produktlinien mit wenigen Produkten weitere Untersuchungen nötig. Eventuell könnte die Produkt-basierte Strategie auch ohne signifikant geringere Verifikationszeit als die Familien-basierte Strategie vorteilhaft sein, da dabei bestehende Model Checker nutzbar sind und beispielsweise keine neuen Werkzeuge und Algorithmen entwickelt werden müssen.

5 Empfehlungen für zukünftige Evaluierungen

In diesem Kapitel werden Empfehlungen und Richtlinien für zukünftige Evaluierungen von Verifikationsänsatzen von Software-Produktlinien erläutert. Während der Untersuchung der Hypothesen sowie der dazugehörigen intensiven Betrachtung der einzelnen Primärstudien wurden Eigenschaften der Evaluierungen erschlossen, die den Vergleich der Ansätze verschiedener Primärstudien erschweren oder vollständig unterbinden. In der folgenden Aufzählung werden darauf basierende Empfehlungen für zukünftige Evaluierungen dargestellt.

1. In dieser Arbeit wurden Hypothesen bezüglich der Verifikationszeit und dem Speicherverbrauch in Abhängigkeit der Anzahl an Features, Produkten und Quellcodezeilen formuliert. Für die Untersuchung dieser Hypothesen müssen folglich die genannten unabhängigen Variablen in den Evaluierungen der einzelnen Primärstudien verfügbar sein. In den verwendeten Primärstudien waren oftmals nur Teile dieser unabhängigen Variablen enthalten, sodass diese Evaluierungsergebnisse nur für eine Teilmenge aller formulierten Hypothesen genutzt werden konnten. Für die Untersuchungen der vorliegenden Arbeit wurden daher zusätzlich die Autoren dieser Primärstudien kontaktiert, um eventuell fehlende Angaben zu den unabhängigen Variablen zu ermitteln, was einen theoretisch vermeidbaren Mehraufwand bedeutet.

   EM1: Auflistung der kompletten Angaben bezüglich der Anzahl an Features, Anzahl an Produkten sowie der Codegröße in Form der Lines of Code der verwendeten Software-Produktlinien.

2. Des Weiteren sollten einheitliche Metriken für die abhängigen Variablen Verifikationszeit und Speicherverbrauch verwendet werden. Beispielsweise wurde die Verifikationszeit größtenteils in üblichen Zeiteinheiten wie Sekunden oder Millisekunden gemessen. Demgegenüber wurde die Verifikationszeit beispielsweise in einer Primärstudie zum Theorem Proving in *Lines of Proof* gemessen und in einer Evaluierung bezüglich des Model Checkings in der Anzahl der Funktionsaufrufe des Verifikationsalgorithmus' evaluiert. Ein ähnliches Bild zeigte sich beim Speicherverbrauch. Der Speicherverbrauch wurde fast ausschließlich in Primärstudien zum Model Checking betrachtet. Dabei wurde der Speicherverbrauch entweder in üblichen Speicherangaben wie Bytes oder in der Anzahl der explorierten Zustände beim Model Checking gemessen. Diese unterschiedlichen Einheiten erschweren den Vergleich der einzelnen Evaluierungen. Basierend darauf wird empfohlen, die Verifikationszeit in Sekunden anzugeben und den Speicherverbrauch in Bytes. Die Anzahl der explorierten Zustände beschränkt sich nur auf den Bereich Model Checking und ermöglicht keine Wiederverwendung

in den anderen Verifikationsstrategien, weshalb er in dieser Arbeit als Empfehlung ausscheidet.

EM2: Verwendung von einheitlichen Metriken für die Verifikationszeit und den Speicherverbrauch. Die Verifikationszeit sollte in Sekunden und der Speicherverbrauch in Bytes gemessen und aufgelistet werden.

3. In den Primärstudien wurde hauptsächlich für das Model Checking der Speicherverbrauch betrachtet und dabei auch lediglich in neun der 28 verwendeten Primärstudien. Besonders beim Model Checking sollte nicht ausschließlich die Verifikationszeit Betrachtung finden, sondern vielmehr sollte der Tradeoff zwischen der Verifikationszeit und des Speicherverbrauchs beim Model Checking berücksichtigt werden. Dieser Tradeoff beschreibt, dass durch eventuelles Senken der Verifikationszeit der Speicherverbrauch erhöht wird und umgekehrt. Jedoch sollte auch bei anderen speicherintensiven Analysen wie Theorem Proving der Speicherverbrauch sowie der Time-Memory-Tradeoff untersucht werden.

   EM3: Bei der Evaluierung sollte vor allem beim Model Checking und anderen speicherintensiven Analysemethoden der Tradeoff zwischen dem Speicherverbrauch und der Verifikationszeit evaluiert werden, anstatt diese abhängigen Variablen separat zu betrachten.

4. Oftmals wurde in den Evaluierungen der Primärstudien eine geringe Anzahl von ein bis drei Software-Produktlinien verwendet. Diese geringe Anzahl an Software-Produktlinien erschweren Aussagen über die Abhängigkeit der Verifikationszeit und des Speicherverbrauchs von der Anzahl an Features, Produkten und Lines of Code. Bestenfalls sollten wie in einigen Primärstudien (z.B. Kolesnikov et al. [65]) viele Software-Produktlinien für die Evaluierungen verwendet werden. Diese Software-Produktlinien sollten alle sowohl Software-Produktlinien mit wenigen sowie vielen Features, Produkten und Codezeilen widerspiegeln. Vor allem war der Anteil an sehr kleinen Software-Produktlinien in den Evaluierungen der Primärstudien sehr gering, was in den Meta-Diagrammen der Hypothesen bezüglich Software-Produktlinien mit wenigen Features, Produkten und Lines of Code deutlich wird. Deshalb sollten für die Evaluierungen zusätzlich deutlich mehr Software-Produktlinien mit wenigen Features, Produkten und Codezeilen verwendet werden. Außerdem sollten auch Software-Produktlinien mit sehr vielen Features, Produkten und Lines of Code bei den Evaluierungen vermehrt genutzt werden. Wünschenswert wäre dazu eine einheitliche Menge quelloffener Software-Produktlinien, die als de facto Standard für die Evaluierungen verwendet werden. Das Projekt SPL2Go[1] verfolgt das Ziel dieses zweiten Aspekts und stellt einen Katalog quelloffener Software-Produktlinien inklusive des Variabilitätsmodells zur Verfügung.

   EM4: Verwendung einer ausreichend großen Menge an Software-Produktlinien, die alle Klassifizierungen für Software-Produktlinien mit wenigen und vielen der betrachteten unabhängigen Variablen abdecken.

5. Ein weiterer Aspekt, der den Vergleich der verschiedenen Ansätze und somit verschiedenen Verifikationsstrategien erschwert ist die Verwendung verschiedener Testumgebungen in

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

Form unterschiedlicher Betriebssysteme, abweichendem Arbeitsspeicher sowie ungleichen Versionen der verwendeten Software. Auch hier wäre die Verwendung einer gleichen Testumgebung vorteilhaft. Dadurch könnten die Evaluierungsergebnisse einfacher miteinander verglichen werden. Jedoch ist diese Empfehlung ebenfalls in der Praxis unrealistisch. Allerdings könnte eine einheitliche Testumgebung mit einer einheitlichen Menge an zur Evaluierung verwendeten Software-Produktlinien einige Vorteile aufweisen. Beispielsweise wäre eine Bestimmung der Break-Even-Points möglich, ab wann welche Verifikationsstrategie vorteilhaft gegenüber einer anderen Strategie ist. Andererseits stellt diese Empfehlung zusätzlich für die Untersuchungen dieser Studie keine große Bedeutung dar, da ausschließlich gepaarte Signifikanztests durchgeführt wurden.

EM5: Evaluierung der zukünftigen Verifikationsansätze mittels einer einheitlichen Testumgebung.

6. Bei der Durchführung der Evaluierungen sollten mehrere Durchläufe jedes Tests durchgeführt werden, um zufällige Einflüsse auszuschließen. Diese Methodik wurde in den meisten der verwendeten Primärstudien verwendet. Jedoch sollten dabei die Evaluierungsergebnisse jedes einzelnen Durchlaufs angegeben werden, ohne die Ergebnisse zusammenzufassen und einen Mittelwert zu berechnen. Dadurch könnte beispielsweise eine Effektgröße aus der Menge der Durchläufe pro Software-Produktlinie und Verifikationsstrategie berechnet werden sowie die Signifikanztest für jeweils eine Software-Produktlinie durchgeführt werden. Außerdem könnten dadurch Signifikanztest für jeden Durchlauf durchgeführt werden, die eventuellen Ausreißer pro Durchlauf identifiziert werden und die Abweichungen pro Durchlauf vom Mittelwert untersucht werden.

EM6: Durchführung mehrerer Evaluierungsdurchläufe sowie Bereitstellung dieser Rohdaten in den Studien oder mittels einer dazugehörigen Datenquelle wie einer Webseite oder einem Filehosting-Dienst.

7. Die unterschiedlichen Verifikationsstrategien bieten des Weiteren verschiedene Möglichkeiten der Evaluierung. Beim Model Checking werden üblicherweise Eigenschaften mittels Aussagenlogik spezifiziert und anschließend evaluiert. Wenn in verschiedenen Primärstudien zum Model Checking dieselben Software-Produktlinien verwendet werden, sollten dabei dieselben Eigenschaften überprüft werden. Dadurch wäre ein direkter Vergleich von zwei Studien möglich. Das Type Checking untersucht die Einhaltung bestimmter Typregeln eines Typsystems. Auch dabei sollten in den Evaluierungen verschiedener Type-Checking-Ansätze anhand derselben Software-Produktlinien die gleichen Typregeln bei der Evaluierung betrachtet werden. Diese beiden Teilempfehlungen ermöglichen einen besseren Vergleich der Model-Checking-Ansätze beziehungsweise Type-Checking-Ansätze untereinander.

EM7: Verifikation einer einheitlichen Menge an Eigenschaftsspezifikationen beim Model Checking und einer einheitlichen Menge an Typregeln beim Type Checking für dieselben Software-Produktlinien.

8. In den meisten Primärstudien wurden die Evaluierungsergebnisse der Verifikationszeit exklusive der benötigten Zeit zur Generierung alle Produkte oder des Metaprogramms angegeben. Diese Zeit sollte jedoch zusätzlich berücksichtigt werden, da sie für die Entscheidung

einer passenden Verifikationsstrategie in der Praxis relevant ist. Die Zeit exklusive der Generierung bietet einen direkten Vergleich des eigentlichen zeitlichen Verifikationsaufwands der jeweiligen Strategie. Dennoch bietet die Zeit inklusive Generierungskosten ebenfalls einen theoretischen als auch praxisrelevanten Vergleich der gesamten Verifikationskosten.

EM8: Evaluierung der Verifikationszeit einer Strategie sowohl inklusive als auch exklusive der zur Generierung der jeweiligen Verifikationskonstrukte benötigten Zeit.

9. In dieser Arbeit wurden die einzelnen Primärstudien darauf untersucht, ob die vorgestellten Strategien jeweils signifikant besser oder schlechter bezüglich der betrachteten abhängigen Variablen ist. Die zukünftigen Evaluierungen sollten diese Signifikanztests bereits enthalten, um potenzielle relevante Verbesserungen durch eine spezifische Strategie zu erkennen.

EM9: Während der Evaluierung von Verifikationsansätzen sollten Signifikanztests durchgeführt werden.

6 Verwandte Arbeiten

Diese Arbeit ist die erste Studie die einen Ansatz-übergreifenden Vergleich der Verifikationsstrategien für formale Verifikationsmethoden von Software-Produktlinien durchführt. Diese Arbeit betrachtet dabei ebenfalls erstmalig die Verifikationsmethoden Model Checking, Type Checking, Static Analysis und Theorem Proving in einer gemeinsamen Arbeit, um Rückschlüsse auf die Vorteile der verschiedenen Verifikationsstrategien herzuleiten. Außerdem werden die Vorteile der Verifikationsstrategien zum ersten Mal in Abhängigkeit von der Anzahl der Produkte, Features und Quellcodezeilen in einer einzigen Arbeit betrachtet.

Als Basis für die vorliegende Studie diente die Arbeit von Thüm et al. [79]. Die vorliegende Arbeit zielt jedoch nicht wie die Studie von Thüm et al. auf die Einordnung der recherchierten Ansätze in die verschiedenen Verifikationsstrategien ab. Dennoch wurden die neu ermittelten Primärstudien ebenfalls in die Klassen dieser Verifikationsstrategien eingeordnet. Außerdem ergab die Überprüfung der einzelnen Primärstudien aus der Arbeit von Thüm et al. keine fehlerhaften Zuordnungen zu den jeweiligen Strategien. Zusätzlich bietet die vorliegende Arbeit eine erste Antwort auf die Vision der Arbeit von Thüm et al., für eine Software-Produktlinie mit spezifischen Eigenschaften die beste Verifikationsstrategie auswählen zu können.

Die in dieser Arbeit verwendeten Primärstudien vergleichen oftmals zwei Verifikationsstrategien miteinander (siehe Anhang A Tabelle A.1.1, A.1.2, A.1.3 und A.1.4). Selten werden mehr als zwei Strategien gegeneinander evaluiert. Somit weisen einige dieser Primärstudien ähnliche Untersuchungsziele wie die vorliegende Arbeit auf. Beispielsweise werden in der Primärstudie von Kolesnikov et al. [65] vier Verifikationsstrategien ausschließlich für einen Type-Checking-Ansatz evaluiert. Apel et al. [71] vergleichen insgesamt fünf Verifikationsstrategien, darunter drei Sampling-basierte Strategien, zum Model Checking von Software-Produktlinien in einer gemeinsamen Studie.

Im Allgemeinen ist der Vergleich von verschiedenen empirischen Evaluierungen im Bereich der Informatik nicht so stark verbreitet wie beispielsweise in der medizinischen Forschung. Dennoch wird in der Studie von Ahnassay et al. [5] ein systematischer Literaturüberblick zu den existierenden empirischen Evaluierungen im Bereich der Software-Produktlinien erstellt. Der Fokus der Arbeit von Ahnassay et al. liegt dabei auf der Beurteilung, welche empirischen Evaluierungen nicht ausreichend geplant oder berichtet wurden.

Im Bereich des Testens existiert zu der vorliegenden Studie eine verwandte Arbeiten. In der Masterarbeit von Paola Accioly [59] werden verschiedene wissenschaftliche Arbeiten vorgestellt, welche zwei unterschiedliche Techniken zur Erzeugung von Testfällen miteinander vergleichen. Die daraus ermittelten Techniken wurden dazu in fünf kontrollierten Experimenten untersucht und die empirischen Ergebnisse zum Vergleich der Produkt-basierten mit einer generischen Technik verwendet.

7 Zusammenfassung, Fazit und Ausblick

In dieser Arbeit wurden die empirischen Evaluierungen verschiedener Primärstudien bezüglich der vorgestellten Verifikationsstrategien in Kombination mit verschiedenen Verifikationsmethoden miteinander verglichen. Dabei wurde sich auf die formalen Verifikationsmethoden Model Checking, Type Checking, Static Analysis und Theorem Proving beschränkt. Die Menge der verwendeten Primärstudien basiert auf der Studie von Thüm et al. [79] und wurde mittels einer systematischen Literaturrecherche um weitere Primärstudien erweitert. Diese Primärstudien stellen verschiedene Verifikationsansätze für Software-Produktlinien vor und evaluieren diese Strategien in empirischen Experimenten und Fallstudien. Anhand dieser empirischen Evaluierungsergebnisse wurden Hypothesen formuliert, die sich auf Vorteile einer bestimmten Verifikationsstrategie gegenüber einer anderen Strategie beziehen. Diese Hypothesen wurden zusätzlich in Verbindung mit der Klassifizierung von wenigen beziehungsweise vielen der betrachteten unabhängigen Variablen formuliert. Dazu wurden die relevanten Daten aus den verwendeten Primärstudien extrahiert und sind im Anhang A dargestellt. Beispielsweise enthält die Tabelle A.1 die unabhängigen Variablen der Software-Produktlinien und die Tabellen zur den Hypothesenuntersuchungen enthalten die abhängigen Variablen. Zum Beispiel ist in Tabelle A.8 die Verifikationszeit der Produkt-basierten und Familien-basierten Strategie dargestellt. Außerdem wurden teilweise zusätzliche Angaben zu unabhängigen Variablen, die in den Primärstudien nicht enthalten waren, ermittelt und in den entsprechenden Tabellen gekennzeichnet. Die Hypothesen wurden zum einen durch die Erzeugung und Interpretation entsprechender Meta-Diagramme und zum anderen mittels Signifikanztests untersucht. Die Evaluierungsergebnisse der Primärstudien, die mindestens zwei unterschiedliche Verifikationsstrategien miteinander vergleichen und die Voraussetzungen des Signifikanztests erfüllen, wurden zusätzlich mittels des Wilcoxon-Vorzeichen-Rang-Tests auf einen signifikanten Unterschied überprüft.

Als Ergebnis zeigte sich in den meisten Primärstudien ein signifikanter Vorteil bezüglich der Verifikationszeit der Familien-basierten Strategie. Vereinzelt wies jedoch die Produkt-basierte Strategie einen geringere Verifikationszeit auf, welche größtenteils jedoch nicht signifikant geringer waren. Ebenfalls wurde bei den meisten untersuchten Hypothesen gezeigt, dass die Familien-basierte Strategie für die entsprechenden Software-Produktlinien mit wenigen beziehungsweise vielen Features, Produkte beziehungsweise Lines of Code signifikant schneller ist. Lediglich bei den verfügbaren Evaluierungen von Software-Produktlinien mit wenigen Produkten ließ sich weder zeigen, dass die Familien-basierte Strategie eine signifikant geringere Verifikationszeit aufweist als die Produkt-basierte Strategie, noch das gegenteilige Verhältnis bestätigen. Damit wird auch die Forschungsfrage F1 teilweise beantwortet, welche Strategie für welche Software-Produktlinien mit welchen spezifischen Eigenschaften besser ist. Zusammenfassend ist die Familien-basierte Strategie somit bes-

ser bezüglich der Verifikationszeit für alle betrachteten Eigenschaften von Software-Produktlinien. Besonders überraschend war dabei das Ergebnis, dass bei Software-Produktlinien mit wenigen Features die Familien-basierte Strategie signifikant schneller ist als die Produkt-basierte Strategie. Außerdem wurden zum einen die Gefahren für die Validität aus den verwendeten Primärstudien zusammengefasst und zum anderen wurden die Gefahren für die Validität der durchgeführten Untersuchungen ausführlich diskutiert.
Des Weiteren wurden anhand der ermittelten Hindernisse und Eigenschaften der bisherigen Evaluierungen einige Empfehlungen und Richtlinien für zukünftige Evaluierungen formuliert. Diese Empfehlungen beziehen sich sowohl auf die Art und Weise der Durchführung der Untersuchungen als auch auf die Evaluierungsergebnisse. Beispielsweise sollten die Evaluierungen immer alle Angaben zur Anzahl der Features, Produkte und der Codezeilen enthalten.

Die Untersuchungen dieser Arbeit stellen lediglich einen ersten Schritt im Vergleich der Verifikationsstrategien dar. Zukünftige Arbeiten könnten die Ansätze der Primärstudien in einer einheitlichen Testumgebung und unter Verwendung einer einheitlichen repräsentativen Menge von Software-Produktlinien erneut evaluieren. Dadurch würde eine bessere Vergleichbarkeit der einzelnen Studien und somit der verschiedenen Verifikationsstrategien gewährleistet. Außerdem ist durch eigene Evaluierungen eine Bestimmung von Break-Event-Points für die unabhängigen Variablen möglich, an denen eine vorher schnellere Verifikationsstrategie nicht mehr schneller als eine andere Strategie ist. Auf Basis dieser ermittelten Schwellenwerte der unabhängigen Variablen könnten genauere Einteilungen in Software-Produktlinien mit wenigen und vielen Features, Produkten und Quellcodezeilen erfolgen.
Darauf aufbauen wären weitere Untersuchungen wie Regressionsanalysen und die Bestimmung weiterer Koeffizienten möglich, womit approximierte Vorhersagen bezüglich der Verifikationszeit einer bestimmten Strategie für eine Software-Produktlinie mit einer gegebenen Anzahl der betrachteten unabhängigen Variablen.
Weiterhin wäre zukünftig eine Bestimmung weiterer beeinflussender Variablen nützlich. In dieser Arbeit wurden ausschließlich die Menge an Features, Produkten und Lines of Code betrachtet. Vor allem die Familien-basierte Strategie profitiert von Gemeinsamkeiten der einzelnen validen Produkte einer Software-Produktlinie. Darauf aufbauend könnten in Zukunft verfeinerte Hypothesen, die den Grad der Gemeinsamkeiten berücksichtigen, formuliert und untersucht werden. Dazu müsste vorweg allerdings ein Maß sowie eine Möglichkeit zur Bestimmung dieses Maßes entwickelt werden.
Bei den Untersuchungen dieser Arbeit wurden alle ermittelten Primärstudien gemeinsam betrachtet. Theoretisch wäre bei ausreichender Anzahl an Primärstudien eine Betrachtung von Untergruppen möglich und bietet bei zunehmender Menge an Primärstudien mit empirischen Evaluierungen eine Grundlage für zukünftige Arbeiten und Untersuchungen. Beispielsweise enthielt die Gesamtmenge an Primärstudien solche Studien, die sich auf evolvierende Software-Produktlinien oder besonderen Eigenschaften wie Multi-Features oder Feature-Attribute fokussieren. Diese Studien bieten die Möglichkeit einer separaten Untersuchung, da sie oftmals die einzigen Studien bezüglich dieser besonderen Eigenschaften sind. So könnte sich ein Ansatz für weiterentwickelte Software-Produktlinien beispielsweise keinen allgemeinen Vorteil der Familien-basierten Strategie aufweisen, wenn die Software-Produktlinie zum ersten Mal verifiziert wird. Dennoch könnte dieser Ansatz

für die Verifikation einer bereits evaluierten Software-Produktlinie nach einer Änderung vorteilhaft sein.
Im Allgemeinen enthielten der Primärstudien hauptsächlich Vergleiche der Produkt-basierten und Familien-basierten Strategie. Zukünftig könnten die Feature-basierte Strategie sowie Kombinationen der drei hauptsächlichen Strategien größere Beachtung finden. Voraussetzung dafür ist allerdings eine Zunahme der Primärstudien für diese drei Strategien, sodass auch die Entwicklung neuer Ansätze basierend auf den kombinierten Strategien zukünftig realisiert werden sollte. Dadurch wäre vor allem eine Untersuchung möglich, ob die kombinierten Strategien ein Einsparungspotenzial bezüglich des Verifikationsaufwands erbringen könnten. Außerdem ist damit nur der Vorteil der Familien-basierten Strategie gezeigt. Für die Feature-basierte sowie die möglichen kombinierten Strategien konnten aufgrund der geringen Anzahl an Primärstudien keine Aussagen getroffen werden. Dadurch ist nicht gewährleistet, dass die Familien-basierte Strategie ebenfalls besser als diese Strategien ist und dass die Familien-basierte Strategie allgemein die effizienteste Strategie für eine gegebene Software-Produktlinie mit spezifischen Eigenschaften ist.

A Anhang A

A.1. Kurzbeschreibung der Ansätze

In den folgenden Abschnitten werden die relevanten Daten der Evaluierungen der Primärstudien vorgestellt. Die Tabellen A.1, A.2, A.3 und A.4 zeigen die unabhängigen Variablen der in den Evaluierungen der Primärstudien verwendeten Software-Produktlinien. Dabei sind die unabhängigen Variablen, die zusätzlich durch Nachfrage gewonnen wurden mit einem Sternsymbol (*) gekennzeichnet. Ein hochgestelltes Pluszeichen ($^{+}$) markiert unabhängige Variablen und Mengen von unabhängigen Variablen, die anhand textueller Beschreibungen ermittelt und mit FeatureIDE rekonstruiert wurden. Zusätzlich sind in diesen Tabellen die evaluierten abhängigen Variablen sowie die verglichenen Verifikationsstrategien aufgelistet. Außerdem wird anschließend an diese Tabellen der Inhalt jeder Primärstudie kurz zusammengefasst.

A.1.1. Model Checking Ansätze

Dokument	Evaluierte SPLs			Was wurde gemessen?	Welche Strategien?
	# Features	# Produkte	# LoC		
	E-Mail Client				
MC-A 1	10	40	-	■ Verifikationszeit	■ PROD ■ FAM
	Künstliche SPL 1				
MC-A 2	7, 9, 11, 13, 15, 17	3, 7, 15, 31, 63, 127, 255	-	■ Verifikationszeit ■ Speed Up	■ PROD ■ FAM
	Künstliche SPL 2				
	4, 7, 10, 13, 16	7, 21, 127, 511, 2047	-		
	Künstliche SPL 3				
	5, 7, 9, 11, 13, 15, 17	5, 7, 9, 11, 13, 15, 17	-		
	Minepump				
MC-A 3	9	64	-	■ Verifikationszeit	■ PROD ■ FAM
	Cash Desk				
MC-A 4	7, 9, 15, 17	9, 18, 27, 54	-	■ Verifikationszeit	■ PROD ■ FAM

MC-A 5	**Coffee Machine**	2, 3, 4, 5, 6, 7*	2, 4, 8, 12, 24, 36+	-	■ Speicherverbrauch (#Zustände) ■ Verifikationszeit ■ Speed Up	■ PROD ■ FAM
MC-A 6	**CFDP**	14	1058	-		■ FAM (normal, für Multi-Features & inkl. numerischer Attribute
	Elevator	5	32	-		
	Minepump	4	9	-		
MC-A 7	**Elevator**	9	512	-	■ Speicherverbrauch (#Zustände)	■ PROD ■ FAM
MC-A 8	**Minepump**	11	128	-	■ Verifikationszeit ■ Speicherverbrauch (#Zustände)	■ PROD ■ FAM
MC-A 9	**KBuild**	4675	-	-	■ Verifikationszeit	■ FAM
	PowerPC Architektur	4495	-	-		
MC-A 10	**E-Mail Client (Java)**	9	40	1233	■ Verifikationszeit	■ PROD ■ SAMPLE ■ FAM
	E-Mail Client (C)	9	40	258		
	Elevator (Java)	6	20	1046		
	Elevator (C)	6	20	877		
	Minepump (Java)	7	64	580		
	Minepump (C)	7	64	279		
	AJStats	20	200	13393		
	GPL	18	42	1405		
	ZipMe	8	10	3636		
MC-A 11	**Minepump**	6, 10, 13	16, 72, 512	-	■ Verifikationszeit	■ PROD ■ FAM

MC-A 12	**Minepump**	4, 9	4, 64	-	■ Verifikationszeit	■ PROD ■ FAM
MC-A 13	**Cash Desk**	4	9^{+}	-	■ Speicherverbrauch (# Zustände)	■ PROD ■ FAM
	Elevator	2	4^{+}	-		
	Vending Machine	5	25^{+}	-		
MC-A 14	**Wiper System**	5	8^{+}	-	■ Verifikationszeit ■ Speicherverbrauch (# Zustände)	■ FAM ■ FAM (mit einer von 3 Reduktionstechniken)
	Vending Machine	8	42^{+}	-		
	Cash Desk	5	18^{+}	-		
	Minepump	6	8^{+}	-		
	Elevator	3	8^{+}	-		
MC-A 15	**Pacemaker**	5^{+}	4	-	■ Verifikationskosten (# Aufrufe von Funktionen)	■ PROD ■ FEAT-PROD
MC-A 16	**Pacemaker**	13	32^{+} (Nur 2 genutzt)	-	■ Verifikationszeit	■ PROD
MC-A 17	**Coffee Machine**	18	118	-	■ Verifikationszeit	■ PROD ■ FAM
MC-A 18	**E-Bond+ Server**	8	39	-	■ Verifikationszeit ■ Speicherverbrauch	■ PROD ■ FAM
MC-A 19	**Minepump**	11*	128	-	■ Speed Up	■ PROD ■ FAM (mit verschiedenen Abstraktionstechniken)
	Elevator	8*	256	-		
	CFDP	10*	56	-		
MC-A 20	**Body Sensor Network**	11	186*	-	■ Verifikationszeit ■ Speicherverbrauch	■ PROD ■ FAM
MC-A 21	**E-Server**	14	17544	-	■ Verifikationszeit ■ Speicherverbrauch	■ PROD ■ FAM

MC-A 22	**Failure-Recovery System** 2, 4, 6, 8, 10, 12, 14, 16 - -	■ Verifikationszeit	■ PROD ■ FAM (2 Techniken)
	Service Provider System 2, 4, 6, 8, 10, 12, 14, 16 - -		
MC-A 23	**Minepump** 4, 7, 9, 11, 12 16, 128, 512, 2048, 4096 -	■ Verifikationszeit ■ Speicherverbrauch (Bytes)	■ FAM (ohne Abstraktion) ■ FAM (mit Abstraktion)
MC-A 24	**Wiper System** 4 4 -	■ Verifikationszeit	■ PROD ■ FAM
	Body Comfort System 6* 21^+ (Nur 5 genutzt) -		
MC-A 25	**SPL Sample Specification** 14 189 -	■ Verifikationszeit	■ FEAT
	SPL Realistic Specification 46 237 -		
MC-A 26	**Vending Machine** 4, 6, 8, 10, 12, 14, 16, 18, 20, 22 9, 18, 36, 72, 144, 288, 576, 1152, 2304, 4608 -	■ Verifikationszeit	■ PROD ■ PROD-OPT
MC-A 27	**Elevator** 9 512 -	■ Verifikationszeit	■ FAM (3 Techniken)
	Telephone SPL 7 128 -		
	Autosoft 10 1024 -		
MC-A 28	**BankAccount** 2, 3, 4, 5, 6, 7, 8, 9, 10 2, 4, 8, 12 ,24, 36, 36, 72, 144 -, -, -, -, -, -, -, -, -, -	■ Verifikationszeit	■ FAM

Tabelle A.1.: Auflistung der Primärstudien zum Model Checking von Software-Produktlinien für den Vergleich der Verifikationsstrategien von Software-Produktlinien

- **Detection of Feature Interactions using Feature-Aware Verification** [74]
  In dem Ansatz von Apel et al. [74] wird sich auf das Erkennen von Feature-Interaktionen konzentriert. Dazu werden die Codebasis und Features als Automaten modelliert. Mit den Schlüsselwörtern *before* und *after* können Ereignisse spezifiziert werden, die sich auf die Zeitpunkte vor bzw. nach einer Methodenausführung beziehen und somit Eigenschaften spezifizieren, die vor bzw. nach einer Methodenausführung gelten sollen. Ein weiteres Schlüsselwort *fail* beschreibt einen Fehlerzustand in Form einer Feature-Interaktion. In diesem Ansatz wird sich auf Eigenschaft-Spezifikationen in LTL fokussiert.

- **Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking** [8]
  In dem Dokument von Classen et al. [8] werden *Featured Transition Systems* (kurz *FTS*) vorgestellt. FTS sind eine Erweiterung von Transitionssystemen 7, welche eine Modellierung der Variabilität von Software-Produktlinien unterstützt. Auf Basis dieser FTS wird in dem Ansatz ein Model Checker namens *SNIP* vorgestellt, der eine Familien-basierte Analyse durch einen in dem Dokument vorgestellten Algorithmus unterstützt.

- **Simulation-Based Abstractions for Software Product-Line Model Checking** [55]
  Der Ansatz von Cordy et al. [55] bietet drei Abstraktionsmethoden, um vor dem Model Checking das zu überprüfende Modell in ein äquivalentes kleineres Modell zu transformieren. Bei der Spezifikation von Eigenschaften wird sich auf CTL fokussiert. Die 1. Abstraktion führt simulations-äquivalente Zustände zusammen. Die 2. Abstraktion hingegen fasst Zustände auf Basis der Erreichbarkeit im Modell zusammen. Für diese Abstraktion wurde bewiesen, dass sie zu einem Modell führt, welches dasselbe Verhalten modelliert wie das Ausgangsmodell. Hingegen zu den Abstraktionen 1 und 2 erzeugt die 3. Abstraktion ein Modell, welches nicht dasselbe Verhalten wie das Ausgangsmodell darstellt. Das abstraktere Modell behält das gesamte Verhalten des Ausgangsmodell bei, fügt jedoch eventuell neues Verhalten hinzu. Die 3. Abstraktion kann zu mehreren unterschiedlichen FTS führen, weshalb der Algorithmus in dem Ansatz von Cordy et al. [55] gierig eines dieser FTS für das Model Checking auswählt.

- **Compositional Algorithmic Verification of Software Product Lines** [33]
  In dem Ansatz von Schaefer et al. [33] schlagen die Autoren ein hierarchisches Modell, genannt *Simple Hierarchical Variability Model* (kurz *SHVM*) zur Beschreibung von einer Menge von Produkten vor. Ein Produkt besteht dabei aus einer Menge von öffentlichen und privaten Methoden. Eine Software-Produktlinie wird in dem Ansatz durch ein Kernmodell, welches bestimmte Erweiterungspunkte besitzt, und weitere SHVMs für Methodenverfeinerungen modelliert. Die Methodenverfeinerungen können mit dem Kernmodell an den Erweiterungspunkten, genannt *Variation Points*, zu einem einzigen Modell zusammengefügt werden. Die Methoden werden in benötigte (*required*) und bereitgestellte (*provided*) unterteilt. Für die Erweiterungspunkte können mehrere unterschiedliche Methodenverfeinerungen existieren. Diese Methodenverfeinerungen müssen die selben benötigten und bereitgestellten Methoden bieten. In dem Ansatz von Schaefer et al. [33] wird sich auf LTL Eigenschaften beschränkt. Der Verifikationsprozess besteht aus zwei Schritten. Im 1. Schritt wird für jede öffentliche Methode der zugehörige Methodengraph extrahiert und mittels *Inlining* der bereits extrahierten Methodengraphen der privaten Methoden zu einem einzelnen Methodengraph transformiert. Die-

ser Methodengraph kann mit einem existierenden herkömmlichen Model Checker analysiert werden. Im 2. Verifikationsschritt werden für alle öffentlichen Methoden die maximalen Methodengraphen und für alle Erweiterungspunkte die maximalen Flussgraphen erzeugt . Diese Graphen werden zu einem Flussgraph zusammengesetzt, für welchen die SHVM-Eigenschaft als LTL-Eigenschaft verifiziert wird.

- **Efficient Verification of Evolving Software Product Lines** [29]
  Sabouri und Khosravi [29] erweitern die Algebra *CCS* um weitere Operatoren, um damit Produktlinien modellieren zu können. Sie nennen diese Produktlinien-Algebra *PL-CCS*. Die zusätzlichen Operatoren in der PL-CCS sind der Varianten-Operator und Optional-Operator zur Modellierung von Erweiterungspunkten. In dem Ansatz von Sabouri und Khosravi [29] wird sich auf evolvierende Software-Produktlinien fokussiert. Der Zustandsraum der Produktfamilie wird einmal generiert. Dieser Zustandsraum wird bei Änderungen der Software-Produktlinie aktualisiert, anstatt dass er komplett neu generiert wird. Bei der Verifikation der sich ändernden Software-Produktlinie werden die bisherigen Verifikationsergebnisse wiederverwendet.

- **Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features** [58]
  Der Ansatz von Cordy et al. [58] ist auf Software-Produktlinien mit numerischen Attributen und Multi-Features ausgerichtet. Numerische Attribute sind nicht-boolesche Features, die je nach Produkt unterschiedliche numerische Werte annehmen können. Multi-Features sind Features, die mehrfach innerhalb des selben Produkts präsent sein können. Zur Modellierung wird *TVL* verwendet, ein textbasierter Ansatz zur Modellierung von Features. Mit TVL werden die validen Produkte einer Software-Produktlinie als valide Kombinationen von Features und Attributwerten modelliert. TVL wird in dem Ansatz von Cordy et al. [58] um die Unterstützung für Multi-Features zu *TVL** erweitert. TVL* wird mit einer Erweiterung für Multi-Features der Sprache Promela, genannt *fPromela*, zur Modellierung des Verhaltens von Software-Produktlinien kombiniert. Als Model Checker wird *SNIP* verwendet, ein Familienbasierter Model Checker auf Basis von FTS.

- **Formal semantics, Modular Specification, and Symbolic Verification of Product-Line Behaviour** [7]
  In dem Ansatz von Classen et al. [7] wird FTS-basiertes Model Checking mit symbolischem Model Checking kombiniert. FTS werden um Annotationen für Transitionen mit booleschen Funktionen erweitert zu FTS^{+}. Mittels FTS+-Modellen wird das komplette Verhalten aller Produkte einer Software-Produktlinie spezifiziert. In diesem Ansatz von Classen et al. [7] wird *fCTL* verwendet. Die Logik fCTL dient zur Spezifikation von Eigenschaften für eine Untermenge aller validen Produkte. Der fCTL-Verifikationsalgorithmus unterteilt den Analysebaum der Formel in einzelne Unterprobleme, die unabhängig behandelt werden können. Es wird der Model Checker *NuSMV* verwendet, welcher als Eingabe eine Liste von Features in der fSVM-Sprache erwartet. Dieser Model Checker ermittelt für jede geprüfte Eigenschaft, welche Produkte die Eigenschaft verletzen oder einhalten.

- **Features meet Scenarios: Modeling and Consistency-Checking Scenario-Based Product Line Specifications** [40]
  Der Ansatz von Greenyer et al. [40] verwendet *modale Sequenzdiagramme* (engl. *modal sequence diagram*, kurz *MSD*) in Kombination mit dem Feature-Diagramm der Software-Produktlinie, um mittels eines Szenario-basierten Ansatzes Interaktionen zwischen Komponenten in einer Software-Produktlinie zu spezifizieren. In den MSDs werden in der Arbeit von Greenyer et al. [40] ausschließlich synchrone Nachrichten verwendet. Bei synchronen Nachrichten bilden das Senden und Empfangen einer Nachricht ein einzelnes Event. Außerdem werden sogenannte Nachrichten-Temperaturen und Ausführungsarten für Nachrichten eingeführt, womit mittels Nachrichten unterschiedliche Spezifikationen von Eigenschaften formuliert werden können. Beispielsweise kann die mit der Ausführungsart *executed* ein Ereignisse spezifiziert werden, das eventuell auftretet muss (*liveness*). Die eigentliche Model Checking-Prozedur basiert auf FTS und verwendet den Model Checker *NuSMV*. Fokussiert wird sich in dem Ansatz von Greenyer et al. [40] auf Eigenschaften, die in CTL spezifiziert sind.

- **Configuration Lifting: Verification meets Software Configuration** [32]
  Der Ansatz von Post und Sinz [32] konvertiert alle Varianten zu einem Metaprogramm, welches zur Verifikation alle Varianten mittels Model Checking verwendet werden kann.

- **Strategies for Product-Line Verification: Case Studies and Experiments** [71]
  Die Arbeit von Apel et al. [71] vergleicht die Produkt-basierte Strategie, eine Sample-basierte Strategie und eine Familien-basierte Strategie für mehrere Software-Produktlinien miteinander.

- **Behavioural Modelling and Verification of Real-Time Software Product Lines** [57]
  Cordy et al. [57] verwenden eine Kombination von Automaten (*Timed Automaton*) und FTS zur Modellierung und Verifikation von Software-Produktlinien für Echtzeit-Systeme. Dazu entwickelten Cordy et al. [57] einen neuen Algorithmus für *Feature Timed Automaton* und fokussieren sich auf zeitbasierte CTL-Logik zur Spezifikation der Eigenschaften.

- **Model Checking lots of systems: Efficient Verification of Temporal Properties in Software Product Lines** [10]
  Der Ansatz von Classen et al. [10] modelliert ein System, welches alle Features enthält. Es wurde ein Werkzeug entwickelt, dass dieses System und somit alle möglichen Produkte gegenüber LTL Eigenschaften verifiziert. Die Eigenschaften werden als Automaten modelliert und das System als FTS. Zur Verifikation wird Überprüft, ob das synchrone Produkt aus FTS und der Negation des Automaten der Eigenschaft eine leere Sprache bilden.

- **Modeling and Verification of Reconfigurable Actor Families** [30]
  In dem Ansatz von Sabouri und Khosravi [30] liegt der Fokus auf der Modellierung und Verifikation von Familien von nebenläufigen Prozessen. Der Ansatz von Sabouri und Khosravi [30] unterstützt dynamische Software-Produktlinie durch eine Möglichkeit zur Rekonfiguration. Modelliert wird die Software-Produktlinie mit *Actor Models* mit der Sprache *Rebeca*. Zur Spezifikation von Eigenschaften wird LTL verwendet.

- **Reducing the verification cost of evolving product families using static analysis techniques** [31]
  Der Fokus des Ansatzes von Sabouri und Khosravi [31] liegt auf der Reduzierung der Verifikationskosten von evolvierenden Software-Produktlinien. Dazu wird versucht, die Anzahl der Produkte, die verifiziert werden sollen, zu verringern. Bei erweiterten Software-Produktlinien, die bereits verifiziert wurden, werden die vorherigen Verifikationsergebnisse wiederverwendet, um die Anzahl zu verifizierender Produkte zu reduzieren. In dem Ansatz von Sabouri und Khosravi [31] werden drei verschiedene Techniken für Model Checking vorgestellt. Die 1. Technik betrachtet Erweiterungen von Software-Produktlinien in Form von hinzugefügten oder entfernten Features und Abhängigkeiten innerhalb des Feature-Modells. In dieser Technik werden nur die neuen Produkte verifiziert. Die 2. und 3. Technik nutzt eine Familienbasierte Strategie. In diesen beiden Techniken wird eine erneute Verifikation von Produkten und Abhängigkeiten, die nicht von der Modifikation der Software-Produktlinie betroffen sind, verhindert. Eigenschaften werden in diesem Ansatz in LTL spezifiziert.

- **Compositional Model Checking of Software Product Lines using Variation Point Obligations** [38]
  Der Ansatz von Liu et al. [38] bietet eine kompositionelle inkrementelle Model Checking Technik. Dabei werden die Features zuerst einzeln analysiert. Die Ergebnisse dieser Analyse werden in späteren Schritten wiederverwendet. Die Codebasis wird als Modell mit Variationspunkten modelliert. Für diese Erweiterungspunkte generiert die kompositionelle Technik sogenannte *Proof Obligations*. Diese *Proof Obligations* bestehen aus einer Menge temporalogischer Formeln, welche die Eigenschaften der Software-Produktlinie repräsentieren. Ein Feature kann an den Erweiterungspunkten an das Basis-System gebunden werden, wenn das Feature die Eigenschaften dieses Erweiterungspunktes nicht verletzt.

- **Towards Formal Safety Analysis in Feature-Oriented Product Line Development** [64]
  Der Ansatz von Bessling und Huhn [64] verknüpft Features mit Fehlermodellen und Safety-Bedingungen. Er bietet eine Modelltransformationstechnik, die sowohl Features zu Produkten transformiert als auch die Fehlermodelle und Safety-Bedingungen in die Produkte integriert. Diese Produkte werden mittels der Modelltransformation als *SysML-Modelle* dargestellt, welche mit dem Werkzeug *SCADE* verifiziert werden.

- **Coherent Branching Feature Bisimulation** [75]
  In der Studie von Belder et al. [75] wird eine Technik vorgestellt, die vor dem Model Checking ein minimales äquivalentes FTS bezüglich eines FTS erzeugt. Das minimale FTS ist minimal bezüglich der Anzahl der Zustände des FTS. Dazu wird eine Technik namens *Coherent Branching Feature Bisimulation* eingesetzt. Das minimale FTS wird anschließend mit dem Model Checker *mCRL2* analysiert.

- **Probabilistic Model Checking for Energy Analyis in Software Product Lines** [26]
  In der Arbeit von Dubslaff et al. [26] wird ein kompositioneller Modellierungsansatz für dynamische Software-Produktlinien vorgestellt, welcher Wahrscheinlichkeits-basiertes Model Checking und die Analyse von quantitativen Eigenschaften unterstützt. Dazu wird der Model Checker *PRISM* in Kombination mit PCTL-Logik verwendet.

- **Counterexample Guided Abstraction Refinement of Product-Line Behavioural Models** [56]
  Der Ansatz von Cordy et al. [56] bietet drei verschiedene Abstraktionsstrategien, um das Modell des zu verifizierenden Systems zu verkleinern. Die *Feature-Abstraction* abstrahiert das Modell in Bezug auf die Variabilität zwischen den Produkten, die *State-Abstraction* nutzt klassische Abstraktionstechniken zur Verringerung der Anzahl der Zustände des Modells. Die dritte Abstraktionsstrategie kombiniert die beiden zuvor beschriebenen Strategien. Des Weiteren werden in diesem Ansatz von Cordy et al. [56] zwei verschiedene Methodiken angewendet, wann das abstraktere Modell verfeinert werden soll. Die Eigenschaften werden dabei in LTL formuliert.

- **Modeling and Verification for Probabilistic Properties in Software Product Lines** [28]
  In der Studie von Rodrigues et al. [28] wird ein Modellierungsansatz für stochastische Software-Produktlinien vorgestellt, welcher *Feature Discrete-Time Marcov Chains* (kurz *FDTMC*) verwendet. Außerdem wird eine neue Verifikationsmethode für stochastische Software-Produktlinien vorgeschlagen. Betrachtet werden zusätzlich zwei existierende Verifikationsmethoden. Die erste Verifikationsmethode transformiert die FDTMCs zu *Discrete-Time Marcov Chains* mit Features als Parametern, welche mit existierenden Model Checkern für parametrisierte DTMCs verifiziert werden können. Die zweite Methode vereinfachte die erste Methode durch Anwendung einer Hierarchie aus zwei Ebenen. Die neue Verifikationstechnik nutzt eine kompakte Struktur, genannt *Feature-Aware Bounded Search* mit einer gebundenen Suche im FDTMC-Modell.

- **Probabilistic Model Checking for Feature-Oriented Systems** [25]
  Der Ansatz von Dubslaff et al. [25] stellt eine simple und intuitive Übersetzung von Software-Produktlinie in die Eingabesprache für den Wahrscheinlichkeits-basierten Model Checker *PRISM*. Es wird eine Familien-basierte Analyse mit symbolischen Methoden zur Reduzierung des Zustandsraums kombiniert. Außerdem wird die dynamische Aktivierung und Deaktivierung von Software-Produktlinien unterstützt.

- **Requirements Verification of Variability-Intensive Systems** [6]
  In der Dissertation von Amir Sharifloo [6] wird sich auf unvollständige Verhaltensspezifikationen und stochastische Eigenschaften fokussiert. Die Variabilität von Software-Produktlinien wird mittels UML Sequenzdiagramme modelliert, welche in FDTMCs transformiert werden. Die Arbeit von Sharifloo [6] betrachtet unter anderem einen enumerativen Ansatz, bei dem FDTMCs zu Markov-Modellen projiziert werden, welche jeweils ein Produkt repräsentieren und separat mittels Model Checking verifiziert wird. Des Weiteren wird ein Methode vorgestellt, bei dem FDTMCs in parametrisierte DTMCs umgewandelt und mit einem existierendem Model Checker für diese DTMCs verifiziert werden. Der dritte Ansatz ist der sogenannte *Bounded*-Ansatz, für den ein neu entwickelter Familien-basierter Model Checker genutzt wird.

- **Family-based Model Checking without a Family-based Model Checker** [2]
  In der Primärstudie von Dimovski [2] werden Abstraktionstechniken vorgestellt, welche die Variabilität in Software-Produktlinien vereinfachen sollen. Des Weiteren wurde eine Technik entwickelt, womit diese abstrakten Variabilitätsmodelle mittels bestehenden Model Checkern wie SPIN analysiert werden können. Der Fokus der Studie von Dimovski et al. [2] liegt auf LTL-

Eigenschaften, die mittels der Variabilität-berücksichtigenden Erweiterung *fLTL* formuliert werden. Außerdem bietet der Ansatz von Dimovski et al. [2] Transformationen von High-Level Modellierungssprachen zu semantischen Modellierungssprachen wie FTS.

- **Incremental Model Checking of Delta-Oriented Software Product Lines** [50]
  In der Primärstudie von Lochau et al. [50] wurde die Algebra *CCS* für eine Verwendung für Delta-orientierte Software-Produktlinien zu *DeltaCCS* erweitert. Dazu wird ein optimierter Produkt-basierter Ansatz zum Model Checking unter Verwendung von DeltaCCS vorgestellt und sowohl mit dem unoptimierten Produkt-basierten Ansatz als auch dem Familien-basierten Ansatz verglichen.

- **Model Checking of Domain Artifacts in Product Line Engineering** [45]
  Der Ansatz von Lauenroth et al. [45] beschreibt eine Möglichkeit zum Model Checking von Domänenartefakten. Dazu werden die Produkte mittels I/O-Automaten spezifiziert und die überprüften Eigenschaften mittels CTL formuliert.

- **Model-Based Verification of Quantitative Non-Functional Properties for Software Product Lines** [15]
  Die Primärstudie von Ghezzi und Sharifloo [15] bietet einen Modell-basierten Ansatz für Wahrscheinlichkeits-basiertes Model Checking von nicht-funktionalen Eigenschaften von Software-Produktlinien.

- **Symbolic Model Checking of Product-Line Requirements using SAT-Based Methods** [68]
  Shoham et al. [68] stellen einen Ansatz zur Adaptierung von Model-Checking-Algorithmen für Software-Produktlinien, die auf dem Erfüllbarkeitsproblem (SAT) basieren.

- **Potential Synergies of Theorem Proving and Model Checking in Software Product Lines** [78]
  In der Studie von Thüm et al. [78] werden die Synergien zwischen Theorem Proving und Model Checking von Software-Produktlinien untersucht. Dazu wird eine Toolchain aus fünf bestehenden Werkzeugen erzeugt, unter anderem wird *MonKey* zum Theorem Proving und *JPF* zum Model Checking verwendet. Der Ansatz von Thüm et al. [78] konzentriert sich dabei hauptsächlich auf Feature-orientierte Software-Produktlinien und Spezifikationen mittels *JML*.

A.1.2. Type Checking Ansätze

Dokument	Evaluierte SPLs			Was wurde gemessen?	Welche Strategien?
	# Features	# Produkte	# LoC		
TC-A 1		**EPL**		■ Verifikationszeit	■ PROD ■ FAM ■ FEAT ■ FEAT-PROD
	12	425	304		
		GPL			
	25	156	2855		
		GraphLib			
	5	16	401		
		GuiDsl			
	26	24	14318		
		Notepad			
	10	512	2193		
		PKJab			
	8	48	4109		
		Prevayler			
	6	32	6185		
		Raroscope			
	4	16	415		
		Sudoku			
	6	64	1926		
		TankWar			
	38	2458	4845		
		Violet			
	88	$\sim 2^{88}$	10866		
		ZipMe			
	13	24	5076		

TC-A 2	**MobileMedia Rel. 2** 2	■ Verifikationszeit	■ FAM
	MobileMedia Rel. 3 4		
	MobileMedia Rel. 4 8		
	MobileMedia Rel. 5 16		
	MobileMedia Rel. 6 48		
	MobileMedia Rel. 7 272		
TC-A 3	**PPL** 7 20 4000	■ Verifikationszeit	■ FAM
	BPL 17 8 28000		
	GPL 18 80 3600		
	JPL 70 56 82000		
TC-A 4	**E-Commerce System** 214 - -	■ Verifikationszeit	■ FAM
TC-A 5	**No SPL** -, -, -, -, -, - 4, 8, 16, 4, 4, 18 -, -, -, -, -, -	■ Verifikationszeit	■ PROD ■ FAM
TC-A 6	**ADT Library** - - 11	■ Verifikationszeit	■ PROD ■ FAM
	Workbench.texteditor - - 16000		
	Workbench.texteditor#2 - - 16000		
	GPL - - 1000		
	ZipMe 14 128 3000		
	Raroscope 5 16 250		
TC-A 7	**Boa (Webserver)** 110 2^{110} 6200	■ Verifikationszeit	■ PROD ■ FAM

TC-A 8	**Busybox**	792	$1{,}26 * 10^{159}$	191615	■ Verifikationszeit ■ Speed Up	■ SAMPLE (3 Strategien) ■ FAM
	Linux	6918	-	6500000		
	OpenSSL	589	$6{,}5 * 10^{175}$	233450		
TC-A 9	**KBuild**	59	~1000000	-	■ Verifikationszeit	■ FAM
TC-A 10	**BusyBox**	811	-	260000	■ Verifikationszeit	■ FAM
TC-A 11	**MobileMedia Rel. 6**	9	144	4600	■ Verifikationszeit	■ PROD ■ FAM
	MobileMedia Rel. 8	14	2784	5700		
	Mobile RSS Reader	14	2048	20000		
	Lampiro	11	2048	45000		
	Berkeley DB	42	$36 * 10^{8}$	70000		
TC-A 12	**KBuild**	4675	-	-	■ Verifikationszeit	■ FAM
	PowerPC Architektur	4495	-	-		

TC-A 13	**EPL**	12	425	126	■ Verifikationszeit	■ PROD ■ FAM ■ FEAT-PROD
	GPL	27	156	2951		
	GraphLib	5	16	596		
	GuiDsl	28	24	15988		
	Notepad	13	512	2732		
	PKJab	8	48	5000		
	Prevayler	6	32	5938		
	Raroscope	5	16	438		
	Sudoku	8	64	2130		
	TankWar	37	2458	5604		
	Violet	89	$\sim 2^{89}$	11006		
	ZipMe	13	24	5096		
TC-A 14	-	-	-	-	■ Verifikationszeit	■ PROD ■ FAM
TC-A 15	**Busybox**	792	$1{,}26 * 10^{159}$	191615	■ Verifikationszeit	■ SAMPLE (3 Strategien) ■ FAM
	Linux	6918	-	6500000		
	OpenSSL	589	$6{,}5 * 10^{175}$	233450		

TC-A 16	**EPL**	12	425	111	■ Verifikationszeit	■ PROD ■ FAM
	GPL	27	156	1930		
	GraphLib	5	16	251		
	GuiDsl	26	24	10084		
	Notepad	12	512	937		
	PKJab	8	48	3373		
	Prevayler	6	32	5268		
	Raroscope	5	16	316		
	Sudoku	7	64	1422		
	TankWar	37	2458	4845		
	Violet	88	-	7151		
	ZipMe	13	24	3446		
TC-A 17	**Busybox**	792	$1{,}26 * 10^{159}$	206815	■ Verifikationszeit	■ SAMPLE (3 Strategien) ■ FAM
	Linux	6918	-	6700000		
TC-A 18	**ZipMe**	17	32	4988	■ Verifikationszeit	■ PROD ■ FAM
	GPL	38	840	2955		
	Notepad	16	1024	2727		

Tabelle A.2.: Auflistung der Primärstudien zum Type Checking von Software-Produktlinien für den Vergleich der Verifikationsstrategien von Software-Produktlinien

- **A Comparison of Product-based, Feature-based, and Family-based Type Checking** [65]
  In der Arbeit von Kolesnikov et al. [65] wurde die Produkt-basierte, die Feature-basierte, die Feature-Produkt-basierte sowie die Familien-basierte Strategie für Type Checking in einem gemeinsamen Werkzeug implementiert. Die implementierte Verifikationsstrategien werden anhand 12 Java-basierter Feature-orientierter Software-Produktlinien verglichen. Type Checking wurde dabei mittels einer Erweiterung des Feature-orientierten Compilers *FUJI* realisiert und der SAT-Solver *SAT4J* wurde zur Erkennung der Abhängigkeiten zwischen Features genutzt.

- **Safe Composition of Configuration Knowledge-based Software Product Lines** [49]
  Die Primärstudie von Teixeira et al. [49] bietet einen Ansatz, bei der Generierung der Produkte eine SPL zu überprüfen, ob alle benötigten Artefakte verfügbar sind. Dazu werden mittels dem Konfigurationswissen (*Configuration Knowledge*) die bereitgestellten und benötigten Schnittstellen beschrieben. Das Feature-Modell beschreibt dabei die Domänenbedingungen und das Konfigurationswissen die Implementierungsbedingungen. Diese beiden Komponenten werden in eine aussagenlogische Formel übersetzt, um die Wohlgeformtheit der SPL zu untersuchen. Die Werkzeuge *Alloy* und *Alloy Analyzer* werden genutzt, um das Feature Modell und das Konfigurationswissen zu codieren, die Verifikation auszuführen und die Liste der nicht-wohlgeformten Produkte zu erzeugen.

- **Safe Composition of Product Lines** [62]
  Thaker et al. [62] stellen einen Ansatz vor, der alle Produkte einer SPL auf Typsicherheit untersucht. Typsicherheit beinhaltet in dem Ansatz von Thaker et al. [62] beispielsweise, dass keine Referenzen zu undefinierten Klassen, Methoden oder Variablen existieren. Der Fokus des Ansatzes von Thaker et al. [62] bezieht sich auf Type Checking unter Verwendung eines SAT-Solvers von Software-Produktlinien, die mittels *AHEAD* implementiert wurden. Im 1. Schritt werden Stubs für alle Klassen und Mitglieder der Klasse wie Methoden oder Felder generiert. Ein Stub wird automatisch generiert und enthält die Signaturen aller Felder, Methoden und möglichen Deklarationen innerhalb der Klasse. Diese Stubs ermöglichen das Erkennen von globalen Konsistenzproblemen. Im 2. Schritt wird ein Type Checking der Feature-Kombinationen ausgeführt. Dabei werden fünf verschiedene Bedingungen betrachtet wie z.B. eine Referenz-Bedingung.

- **Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints** [47]
  Der Ansatz von Czarnecki und Pietroszek [47] bietet einen automatisierten Verifikationsprozess zur Überprüfung, dass keine fehlerhaften Template-Instanzen durch korrekte Featurekonfiguration generiert werden können. In dieser Studie von Czarnecki und Pietroszek [47] werden Feature-basierte Modell-Templates verwendet, welche aus einem Feature Modell und einem annotierten Modell in einer allgemeinen Modellierungsprache wie beispielsweise UML. Bei der Verifikation werden nur *Prescence Conditions* berücksichtigt, welche analog zu *#ifdef* -Direktiven sind. Mittels dieser Prescence Conditions werden die Elemente eines Modells annotiert, die in einer Template-Instanz vorhanden und nicht vorhanden sein sollen. Die Wohlgeformtheitseigenschaften der SPL werden in der Arbeit von Czarnecki und Pietroszek [47] mittels der *Object Constraint Language (OCL)* formuliert.

- **Extending Type Inference to Variational Programs** [66]
  Die Primärstudie von Chen et al. [66] stellt das *variable Lambda-Kalkül (VLC)* für statische Analysen von variablen Programmen vor. VLC erweitert dazu das Lambda-Kalkül um Konstrukte zur Spezifikation und Organisation von statischer Variabilität. Des Weiteren wird ein Typsystem für das VLC vorgestellt, welches variable Programme zu variablen Typen zugewiesen werden können.

- **Safe Composition of Non-Monotonic Features** [52]
  Kuhlemann et al. [52] beschreiben eine Möglichkeit zur sicheren Komposition (*Safe Composition*) von nichtmonotonen Features. Monotone Features erlauben das Hinzufügen von Feldern und Methoden sowie das Wrapping von Methoden. Nichtmonotone Features ermöglichen zusätzlich das Erzeugen und Löschen von Codefragmenten. Diese Technik betrachtet sogenannte *Refactorings*. Refactorings sind Programmtransformationen, welche die Struktur des Programms ohne Beeinflussung der Semantik ändern. Dazu werden Elemente, die vor dem Refactoring verfügbar sein müssen und nicht verfügbar sein dürfen, durch Vorbedingungen spezifiziert. Der Ansatz von Kuhlemann et al. [52] berechnet eine Menge von Feature-Kompositionen, die bestimmte Elemente benötigen beziehungsweise ausschließen.

- **TypeChef: Toward Type Checking #ifdef Variability in C** [11]
  In der Studie von Kenner et al. [11] werden als erstes die Schwierigkeiten beim Type Checking von C-Code erläutert. Anschließend wird eine Technik zum Type Checking von unprozessierten C-Code mittels TypeChef vorgestellt und implementiert. TypeChef kommentiert im ersten Schritt alle #ifdef-Anweisungen mittels einem partiellen Präprozessor aus, sodass nur noch #define- und #include-Anweisungen sichtbar sind. Im zweiten Schritt werden die #ifdef-Anweisungen zu disziplinierten Annotationen umformuliert. Disziplinierte Annotationen umschließen ausschließlich komplette Methoden oder Anweisungen. Der dritte Schritt beinhaltet die Generierung eines Parsers für TypeChef anhand einer definierten Grammatik. Im vierten Schritt werden die Referenzen ermittelt, die überprüft werden sollen. Dabei wird eine Menge von Formeln erzeugt, die als Eingabe des Solving-Werkzeugs dienen. Im fünften Schritt erfolgt die Lösung dieser Formeln.

- **Scalable Analysis of variable Software** [43]
  Der Ansatz von Liebig et al. [43] vergleicht Familien-basierten Type-Checking mit optimierten Produkt-basierten Type Checking basierend auf vier verschiedenen Sampling-Strategien anhand von drei großen realen SPLs.

- **Supporting Consistency Checking between Features and Software Product Line Use Scenarios** [54]
  Alférez et al. [54] stellen eine Konsistenzprüfung zwischen Features und Use Scenarios in Form von UML Use Case- und Aktivitätsdiagrammen dar. Dabei wird überprüft, ob in anderen Varianten definierte Elementen in anderen Varianten verfügbar sind.

- **A Variability-Aware Module System** [19]
  In der Studie von Kästner et al. [19] wird ein Modulsystem vorgestellt, dass Kompilierungszeitvariabilität innerhalb von Modulen und dessen Schnittstelen erlaubt. Kästner et al. [19]

behaupten, dass dieser Ansatz einen ersten Schritt auf dem Weg zu *Produktlinien von Produktlinien* darstellt. Die Variabilität innerhalb von Modulen sowie die Verlinkung von Variablen auf Kompositionsebene wird in diesem Modulsystem mittels #ifdef-Annotationen realisiert.

- **Type Checking Annotation-Based Product Lines** [20]
  Ein Beitrag der Primärstudie von Kästner et al. [20] ist die Erläuterung der Typing-Probleme für Software-Produktlinien. Außerdem wird ein Typsystem für Software-Produktlinien sowie eine Generierungstechnik für die Varianten auf Basis von *Featherweight Java* formalisiert. Zusätzlich wird der Type Checker für Java in das Werkzeug *CIDE* implementiert.

- **Configuration Lifting: Verification meets Software Configuration** [32]
  In dem Ansatz von Post und Sinz [32] werden alle Varianten einer Software-Produktlinie zu einem Meta-Programm transformiert, welches zur Verifikation alle Varianten mittels Type Checking verwendet werden kann.

- **Experiments on Type Checking of Software Product Lines** [24]
  In der Masterarbeit von Claus Hunsen [24] wird ein Feature-Produkt-basierter Ansatz vorgestellt, welcher die Feature-Kompilierung und das Type Checking auf Bytecode-Ebene ausführt. Dieser Ansatz wird für 12 Java-basierte Software-Produktlinien mit der Produkt-basierten und der Familien-basierten Strategie verglichen.

- **Type-based Parametric Analysis of Program Families** [67]
  Die Primärstudie von Chen und Erwig [67] beschreibt ein parametrisiertes Framework für statische Analysen für Programmfamilien. Dieses Framework bietet ein automatisches Lifting von Typ-basierten statischen Analysen einzelner Programme zu Analysen für Software-Produktlinien. In der Studie von Chen und Erwig [67] werden hauptsächlich Annotations-basierte Software-Produktlinien betrachtet.

- **Analysis and Transformation of Configurable Systems** [41]
  Ein Teilbeitrag der Doktorarbeit von Jörg Liebig [41] ist der Vergleich zwischen Familien-basierten Type Checking und drei häufig verwendeten Sample-basierten Strategien in Verbindung mit Type Checking für drei große reale Software-Produktlinien.

- **Typprüfung von Produktlinien in FUJI** [61]
  In der Masterarbeit von Peter Lutz [61] wird ein Familien-basierter Type Checker für Feature-orientierte Software-Produktlinien für den Compiler *FUJI* entwickelt. Anschließend wird dieser Type Checker mit der Produkt-basierten Strategie anhand von zwölf Software-Produktlinien verglichen.

- **Large-Scale Variability Aware Type Checking and Dataflow Analysis** [42]
  Der Ansatz von Liebig et al. [42] beschreibt einen Familien-basierten Type Checker und Datenfluss-Analysierer für C-basierte Software-Produktlinien mit Präprozessor-Anweisungen. Die vorgestellte Familien-basierte Strategie wird mit drei gebräuchlichen Sampling-Strategien in Kombination mit Type Checking dieser ausgewählten Produkte für zwei reale SPLs verglichen.

- **Typsicherheit in Feature-orientierten Software-Produktlinien in FeatureIDE** [69]
  Die Arbeit von Sönke Holthusen [69] stellt die Implementierung eines Familien-basierten Type Checkers für Feature-orientierte Software-Produktlinien für die Entwicklungsumgebung *FeatureIDE* vor.

A.1.3. Static Analysis Ansätze

Dokument	Evaluierte SPLs			Was wurde gemessen?	Welche Strategien?
	# Features	# Produkte	# LoC		
SA-A 1	**Graph PL**			■ Verifikationszeit ■ Speicherverbrauch	■ PROD ■ FAM (4 Ansätze)
	18	512	1350		
	MobileMedia Release 8				
	14	128	5700		
	Prevayler				
	5	8	8000		
	Lamprio				
	11	4	45000		
	BerkeleyDB				
	42	128	84000		
SA-A 2	**BerkeleyDB**			■ Verifikationszeit	■ FAM (2 Ansätze)
	56	$5 * 10^{10}$	84000		
	GPL				
	29	524288	1400		
	Lampiro				
	20	4	45000		
	MobileMedia Release 8				
	34	512	5700		
SA-A 3	**SCARI**			■ Verifikationszeit	■ FAM
	14	-	257		
	GPL				
	13	-	590		
SA-A 4	**GPL**			■ Verifikationszeit	■ FAM
	18	156	1713		
	Notepad				
	25	7057	2074		
	jak2java				
	17	5	26332		
SA-A 5	**GPL**			■ Verifikationszeit	■ FAM
	18	156	1713		
	Notepad				
	25	144	2074		

SA-A 6	**Wiper System**	5	8	-	■ Verifikationszeit ■ Speicherverbrauch (# Zustände)	■ FAM ■ FAM (mit einer von 3 Reduktionstechniken)
	Vending Machine	8	42	-		
	Cash Desk	5	18	-		
	Minepump	6	8	-		
	Elevator	3	8	-		
SA-A 7	**GPL::V'.display()**	-	106	-	■ Verifikationszeit ■ Speed Up	■ PROD ■ FAM (2 Ansätze)
	GPL::G'.run()	-	72	-		
	BerkeleyDB::D'.main()	-	40			
	Prevayler::P'.publisher()	-	8	-		
SA-A 8	**Prevayler::P'F'.publisher()**	5	-	8000	■ Verifikationszeit	■ FAM (4 Ansätze)
	BerkeleyDB:: DBRunAction.main()	42	-	84000		
	GPL::Vertex.display()	18	-	1350		
SA-A 9	**Busybox**	792	$1{,}26 * 10^{159}$	191615	■ Verifikationszeit	■ SAMPLE (3 Strategien) ■ FAM
	Linux	6918	-	6500000		
	OpenSSL	589	$6{,}5 * 10^{175}$	233450		
SA-A 10	**Busybox**	792	$1{,}26 * 10^{159}$	206815	■ Verifikationszeit	■ SAMPLE (3 Strategien) ■ FAM
	Linux	6918	-	6700000		

SA-A 11	792	**Busybox** $1{,}26 * 10^{159}$	191615	■ Verifikationszeit ■ Speed Up	■ SAMPLE (3 Strategien) ■ FAM
	6918	**Linux** -	6500000		
	589	**OpenSSL** $6{,}5 * 10^{175}$	233450		

Tabelle A.3.: Auflistung der Static-Analysis-Primärstudien von Software-Produktlinien für den Vergleich der Verifikationsstrategien von Software-Produktlinien

- **Intraprocedural Dataflow Analysis for Software Product Lines** [23]
  Der Ansatz von Brabrand et al. [23] bietet fünf verschiedene Techniken zur Umwandlung jeder intraprozeduralen Datenflussanalyse in eine Feature-sensitive Datenflussanalyse für Software-Produktlinien. Die erste Technik *A1* bietet eine Möglichkeit zur Repräsentation der Variabilität im Kontrollflussgraph sowie in den Transferfunktionen, wobei die Konfigurationen trotzdem einzeln nacheinander analysiert werden. Die zweite Methode *A2* ermöglicht die gleichzeitige Analyse aller Konfigurationen in einem Durchgang. Die dritte Technik A3 erweitert den Ansatz A2 um ein Verfahren zur Informationsteilung zwischen den einzelnen Konfigurationen während der Analyse. Das Verfahren *A4* abstrahiert den Ansatz A3, sodass äquivalente Konfigurationen während des Analysedurchgangs zusammengeführt werden. Die fünfte Technik *A5* stellt die Kombination der Methoden A1 bis A4 dar. Implementiert wurden alle Ansätze in das Framework *SOOT* zur intraprozeduralen Datenflussanalyse.

- **SPL^{LIFT}: Statically Analyzing Software Product Lines in Minutes instead of Years** [27]
  In der Studie von Bodden et al. [27] wird ein Ansatz zur automatischen Konvertierung von Analysen für einzelne Programme innerhalb des IFDS Frameworks für interprozedurale Datenflussanalysen für Software-Produktlinien vorgestellt. Dazu wurde der Solver *Heros* entwickelt und eine Toolchain aus Heros, *SOOT*, *CIDE* und *JavaBDD* implementiert.

- **Integration Testing of SPLs Using Compositional Symbolic Execution** [37]
  Der Ansatz von Shi et al. [37] bietet eine auf statischer Analyse basierende Technik zur Verringerung der Menge zu überprüfenden Produkten während des Testens. Dabei wird mittels symbolischer Ausführung aus einem Feature-Dependence-Graph eine Menge von Interaktionsbäumen erzeugt.

- **Reducing Combinatorics in Testing Product Lines** [16]
  Diese Primärstudie von Kim et al. [16] stellt einen Ansatz vor, der die Menge an für einen dynamischen Test notwendigen Produkte reduziert. Unter Verwendung einer statischen Analysemethode werden irrelevante Features ermittelt, welche das Ergebnis eines Tests nicht beeinflussen.

- **Reducing Configurations to Monitor in a Software Product Line** [17]
  Der Ansatz von Kim et al. [17] kombiniert statische Analyse mit Testen. Die statische Analyse ermittelt die Produkte einer Software-Produktlinie, welche eine spezifizierte Safety-Eigenschaft nicht verletzen können. Somit wird eine Teilmenge aller validen Produkte einer Software-Produktlinie bestimmt, die mittels Runtime Monitoring für eine spezifische Safety-Eigenschaft kontrolliert werden müssen.

- **Reducing the Verification Cost of Evolving Product Families using Static Analysis Techniques** [31]
  In dem Ansatz von Sabouri und Khosravi [31] wird die statische Analyse zur Reduzierung der zu verifizierenden Produkte verwendet. Der Fokus dieser Reduzierung liegt auf sich ändernden Software-Produktlinien. Eine Wiederverwendung von bisherigen Verifikationsergebnissen vor der Evolution der Software-Produktlinie verringert die Anzahl an nach der Evolution zu verifizierenden Produkte.

- **Systematic Derivation of Correct Variability-Aware Program Analyses** [34]
  Der Ansatz von Midtgaard et al. [34] stellt eine Technik namens *Variational Abstract Interpretation* vor. Diese Technik ermöglicht die automatische Transformation von statischen Analysen, welche auf abstrakter Interpretation basieren, zu Variabilität-berücksichtigende Analysen für Software-Produktlinien. In der enthaltenen Fallstudie wird mittels dieser Technik eine Datenflussanalyse für *Reaching Definitions* im intraprozeduralen Analyseframework *SOOT* zu einer äquivalenten Familien-basierten Datenflussanalyse für Software-Produktlinien transformiert und evaluiert.

- **Variability Abstractions: Trading Precision for Speed in Family-Based Analyses** [3]
  Die Studie von Dimovski et al. [3] bietet verschiedene Techniken zur Abstraktion der Variabilität von Software-Produktlinien, um daraus abstrakte Analysen für Software-Produktlinien herzuleiten. Die Abstraktionstechniken basieren dabei auf *Galois-Verbindungen* in Kombination mit abstrakter Interpretation.

- **Analysis and Transformation of Configurable Systems** [41]
  In dieser Doktorarbeit von Jörg Liebig [41] wird unter anderem die Datenflussanalyse basierend auf der Familien-basierten Strategie mit drei Sampling-basierten Strategien für die Datenflussanalyse verglichen.

- **Large-Scale Variability-Aware Type Checking and Dataflow Analysis** [42]
  In dem Ansatz von Liebig et al. [42] wird ein Variabilität-berücksichtigender Type Checker und Datenfluss-Analysierer für mittels der Programmiersprache C implementierte Präprozessor-basierte Software-Produktlinien und ein entsprechendes Build-System realisiert.

- **Scalable Analysis of Variable Software** [43]
  In der Primärstudie von Liebig et al. [43] wird eine Familien-basierte Datenflussanalyse mit vier Sample-basierten Heuristiken für drei große Software-Produktlinien verglichen.

A.1.4. Theorem Proving Ansätze

Dokument	Evaluierte SPLs			Was wurde gemessen?	Welche Strategien?
	# Features	# Produkte	# LoC		
TP-A 1	**BankAccount**			■ Verifikationszeit	■ PROD ■ FAM
	2, 3, 4, 5, 6	2, 4, 8, 12 ,24	75, 86, 124, 135, 140		
TP-A 2	**BankAccount**			■ Verifikationszeit (Lines of Proof)	■ PROD ■ FEAT-PROD
	5	24	140		
TP-A 3	**Feature Featherweight Java**			■ Verifikationszeit	■ FAM
	6	-	12209		
TP-A 4	**BankAccount**			■ Verifikationszeit	■ FAM (versch. Abstaktionen)
	8	72	239		
TP-A 5	**BankAccount**			■ Verifikationszeit	■ PROD ■ FAM
	1, 2, 3, 4, 5, 6, 7, 8	1, 2, 4, 8, 12 ,24, 48, 72	51, 57, 95, 106, 117, 172, 198, 239		
TP-A 6	**BankAccount**			■ Verifikationszeit	■ FAM
	2, 3, 4, 5, 6, 7, 8, 9, 10	2, 4, 8, 12 ,24, 36, 36, 72, 144	-, -, -, -, -, -, -, -, -		

Tabelle A.4.: Auflistung der Primärstudien zum Theorem Proving von Software-Produktlinien für den Vergleich der Verifikationsstrategien von Software-Produktlinien

- **Family-Based Deductive Verification of Software Product Lines** [77]
  In der Arbeit von Thüm et al. [77] wird ein Familien-basierter Ansatz zur deduktiven Verifikation von Feature-orientierten Software-Produktlinien vorgestellt. Dazu wird für die Software-Produktlinie sowohl eine Metaprodukt als auch eine Metaspezifikation erzeugt. Diese beiden Metakonstrukte werden mittels Kontrakten in der *Java Modelling Language (JML)* und dem Theorem Prover *KeY* verifiziert.

- **Proof Composition for Deductive Verification of Software Product Lines** [76]
  Der Ansatz von Thüm et al. [76] präsentiert eine Technik namens *Proof Composition*, mit welcher aus Beweisen einzelner Features komplette Beweise für eine Software-Produktlinie erzeugt werden können. Der Fokus liegt dabei auf Spezifikationen mittels *Design by Contract* mit JML und Feature-orientierten Software-Produktlinien. Zur Implementierung dieser Technik wurde eine Toolchain aus *Krakatoa/Why* mit dem Beweisassistenten *Coq* kombiniert.

- **Product Lines of Theorems** [13]
  Die Primärstudie von Delaware et al. [13] bietet einen Ansatz, um aus Beweisfragmenten einzelner Features Korrektheitsbeweise für Produkte mit diesem Feature zu erzeugen. Dazu werden Feature-Module mit Beweisfragmenten angereichert, welche durch Kombination Theoreme und Beweise für die gesamte Software-Produktlinie erzeugen können. Die Implementierung dieser Technik erfolgte in den Beweisassistenten *Coq*.

- **Effiziente Kodierung von Variabilität in Spezifikationen** [53]
  Die Masterarbeit von Matthias Praast [53] präsentiert neue Kompositionsverfahren für Spezifikationen in Form von Kontrakten. Dabei werden die komponierten Spezifikationen vereinfacht, um zusätzlich eine erhöhte Lesbarkeit der Spezifikation zu erreichen. Des Weiteren wird die Auswirkung dieser Vereinfachungen auf Familien-basiertes Theorem Proving untersucht.

- **JML-Based Verification for Feature-Oriented Programming** [36]
  In der Bachelorarbeit von Jens Meinicke [36] wird eine Werkzeug-Unterstützung zur automatisierten Generierung eines Metaprodukts für Feature-orientierte Software-Produktlinien entwickelt und implementiert sowie anschließend bezüglich der Performanz evaluiert.

- **Potential Synergies of Theorem Proving and Model Checking in Software Product Lines** [78]
  In der Studie von Thüm et al. [78] werden die Synergien zwischen Theorem Proving und Model Checking von Software-Produktlinien untersucht. Dazu wird eine Toolchain aus fünf bestehenden Werkzeugen erzeugt, unter anderem wird *MonKey* zum Theorem Proving und *JPF* zum Model Checking verwendet. Der Ansatz von Thüm et al. [78] konzentriert sich dabei hauptsächlich auf Feature-orientierte Software-Produktlinien und Spezifikationen mittels *JML*.

A.2. Tabellen

In diesem Abschnitt werden die Tabellen mit den Ergebnissen der Signifikanztests und der Berechnung der standardisierten Effektgröße d_{Cohen} vorgestellt. Zusätzlich sind die Tabellen aufgeführt, welche die Verifikationszeiten und den Speicherverbrauch der verglichenen Verifikationsstrategien bezüglich der Hypothesen enthalten.

A.2.1. Tabellen zum Model Checking

Die nachfolgenden Tabellen enthalten die Datensätze, die für die Untersuchung der Hypothesen zum Model Checking von Software-Produktlinien verwendet wurden. Zuerst werden in der Tabelle A.5 die Werte des Wilcoxon-Vorzeichen-Rang-Tests für die Tabelle 4.10 aufgelistet. Die Tabelle A.7 enthält die berechneten Daten für die Effektgröße d_{Cohen}. Anschließend stellen die Tabellen A.8 bis A.14 die jeweilige unabhängige und abhängige Variable der Hypothesenuntersuchung jeder verwendbaren Primärstudie inklusive der entsprechenden abhängigen Variable dar. Diese Tabellen enthalten somit ebenfalls die Daten der Meta-Diagramme des Model Checkings.

Strategie A	**Strategie B**	**SPL**	w	n	$w_{\alpha,n}$	p**-Wert**
SINGLE-WISE	PROD	E-Mail Java	0	9	8	0,00195
SINGLE-WISE	PROD	E-Mail C	0	10	10	0,000977

SINGLE-WISE	PROD	Elevator Java	0	9	8	0,00195
SINGLE-WISE	PROD	Elevator C	0	6	2	0,0156
SINGLE-WISE	PROD	Minepump Java	0	5	0	0,00313
SINGLE-WISE	PROD	Minepump C	0	5	0	0,00313
PAIR-WISE	PROD	E-Mail Java	0	9	8	0,00195
PAIR-WISE	PROD	E-Mail C	0	10	10	0,000977
PAIR-WISE	PROD	Elevator Java	0	9	8	0,00195
PAIR-WISE	PROD	Elevator C	0	6	2	0,0156
PAIR-WISE	PROD	Minepump Java	0	5	0	0,00313
PAIR-WISE	PROD	Minepump C	0	5	0	0,00313
TRIPLE-WISE	PROD	E-Mail Java	0	9	8	~0,00455
TRIPLE-WISE	PROD	E-Mail C	0	10	10	~0,00295
TRIPLE-WISE	PROD	Elevator Java	0	9	8	0,00195
TRIPLE-WISE	PROD	Elevator C	0	6	2	0,0295
TRIPLE-WISE	PROD	Minepump Java	0	5	0	0,00313
TRIPLE-WISE	PROD	Minepump C	0	5	0	0,00313
FAM	SINGLE-WISE	E-Mail Java	17	9	8	0,285
FAM	SINGLE-WISE	E-Mail C	55	10	10	1
FAM	SINGLE-WISE	Elevator Java	44	9	8	0,998
FAM	SINGLE-WISE	Elevator C	0	6	2	0,0156
FAM	SINGLE-WISE	Minepump Java	0	5	0	0,00313
FAM	SINGLE-WISE	Minepump C	0	5	0	0,00313
FAM	PAIR-WISE	E-Mail Java	17	9	8	0,285
FAM	PAIR-WISE	E-Mail C	55	10	10	1
FAM	PAIR-WISE	Elevator Java	0	9	8	0,00195
FAM	PAIR-WISE	Elevator C	0	6	2	0,0156
FAM	PAIR-WISE	Minepump Java	0	5	0	0,00313
FAM	PAIR-WISE	Minepump C	0	5	0	0,00313
FAM	TRIPLE-WISE	E-Mail Java	0	9	8	0,00195
FAM	TRIPLE-WISE	E-Mail C	24,5	10	10	~0,399
FAM	TRIPLE-WISE	Elevator Java	0	9	8	0,00195
FAM	TRIPLE-WISE	Elevator C	0	6	2	0,0156
FAM	TRIPLE-WISE	Minepump Java	0	5	0	0,00313
FAM	TRIPLE-WISE	Minepump C	0	5	0	0,00313
SINGLE-WISE	FAM	E-Mail Java	28	9	8	0,752
SINGLE-WISE	FAM	E-Mail C	0	10	10	0,000977
SINGLE-WISE	FAM	Elevator Java	1	9	8	0,00391
SINGLE-WISE	FAM	Elevator C	21	6	2	1
SINGLE-WISE	FAM	Minepump Java	15	5	0	1
SINGLE-WISE	FAM	Minepump C	15	5	0	1
PAIR-WISE	FAM	E-Mail Java	28	9	8	0,752
PAIR-WISE	FAM	E-Mail C	0	10	10	0,000977
PAIR-WISE	FAM	Elevator Java	45	9	8	1
PAIR-WISE	FAM	Elevator C	21	6	2	1

PAIR-WISE	FAM	Minepump Java	15	5	0	1
PAIR-WISE	FAM	Minepump C	15	5	0	1
TRIPLE-WISE	FAM	E-Mail Java	45	9	8	1
TRIPLE-WISE	FAM	E-Mail C	30,5	10	10	0,639
TRIPLE-WISE	FAM	Elevator Java	45	9	8	1
TRIPLE-WISE	FAM	Elevator C	21	6	2	1
TRIPLE-WISE	FAM	Minepump Java	15	5	0	1
TRIPLE-WISE	FAM	Minepump C	15	5	0	1

Tabelle A.5.: Ergebnisse des Wilcoxon-Vorzeichen-Rang-Tests für die Verifikationszeit der verglichenen Strategien der Primärstudie MC-A 10

In der Tabelle A.7 ist die entsprechende Studie aufgelistet, welche die Evaluierung enthält, sowie die Mittelwerte und Standardabweichungen für die Produkt-basierte und die Familien-basierte Strategie. Des Weiteren ist die berechnete Effektgröße, der Stichprobenumfang n und das 95%ige Konfidenzintervall für die den Wert von d_{Cohen} und der Stichprobengröße dargestellt.

Studie	SPL	Mittelwert PROD	Standardabweichung PROD	Mittelwert FAM	Standardabweichung FAM	d_{Cohen}	n	95% Konfidenzintervall
MC-A 2	Binary SPL	1517	3686	1266	3116	-0,074	8	[-1,46 ; 1,31]
MC-A 2	Ternary SPL	1490	3124	280,3	583	-0,538	8	[-1,95 ; 0,872]
MC-A 2	Unary SPL	765	1239	380	470	-0,411	8	[-1,81 ; 0,989]
MC-A 3	Minepump	40.4	45,1	13,1	27,7	-0,729	10	[-2,01 ; 0.551]
MC-A 7	Elevator	18,3	2,15	1,72	2,28	-7,48	15	[-10,3 ; -4,62]
MC-A 7	Elevator	32,9	5.83	4,42	4,18	-5,62	15	[-7,87 ; -3,37]
MC-A 7	Elevator	66,9	28	40,8	81,5	-0,429	15	[-1,45 ; 0,595]
MC-A 7	Elevator	185	156	106	154	-0,506	15	[-1,54 ; 0,522]
MC-A 8	Minepump	6,01	5,02	1,19	1,61	-1,28	45	[-1,92 ; 0,639]
MC-A 10	E-Mail Java	36,3	26,1	16,6	13,3	-0,95	9	[-2,33 ; 0,428]
MC-A 10	E-Mail C	219	155	170	103	-0,373	10	[-1,62 ; 0,877]
MC-A 10	Elevator Java	79,4	31,2	42	17,1	-1,49	9	[-2,96 ; 0,01]
MC-A 10	Elevator C	36,4	15,4	4,16	1,24	-2,95	6	[-5,26 ; -0,636]
MC-A 10	Minepump Java	177	54,9	8,34	1,19	-4,36	5	[-7,58 ; -1,14]
MC-A 10	Minepump C	104	3,55	3,57	0,379	-39,9	5	[-64,7 ; -15,1]
MC-A 10	AJStats	214	59,4	29,6	3,39	-4,38	2	[-9,5 ; 0,729]
MC-A 10	GPL	39,6	1,7	2,52	0,24	-30,6	2	[-60,7 ; -0,484]
MC-A 11	Minepump	0,534	0,225	1,06	0,875	0,818	12	[-0,36 ; 2,0]
MC-A 11	Minepump	5,2	1,95	2,44	2,85	-1,13	12	[-2,35 ; 0,087]
MC-A 11	Minepump	14,9	7,06	2,72	3,02	-2,24	12	[-3,68 ; -0,795]
MC-A 12	Minepump	9,3	8,23	10,1	13,5	0,071	6	[-1,53 ; 1,67]
MC-A 12	Minepump	43,3	49,8	21,8	39,6	-0,476	6	[-2,1 ; 1,15]
MC-A 17	Coffee Machine	42,1	0,911	62,9	23,4	1,26	12	[0,017 ; 2,49]
MC-A 17	Coffee Machine	38,5	1,09	17,4	13,1	-2,27	12	[-3,73 ; -0,821]
MC-A 20	BSN	68,7	89,2	2,97	8,17	-1,04	9	[-2,43 ; 0,353]
MC-A 22	Service Provider	115	160	1,06	1,01	-1,01	8	[-2,48 ; 0,461]
MC-A 22	Failure Recovery	112	156	23,1	36,6	-0,791	8	[-2,23 ; 0,648]

Tabelle A.7.: PROD & FAM - Mittelwerte und Standardabweichungen der einzelnen Primärstudien, Effektstärke d_{Cohen} & das 95 % Konfidenzintervallen

Dokument	#Produkte	Zeit(s) PROD	Zeit(s) FAM	SPL
MC-A 2	9	3,07	3,04	Abstrakte SPL Unary
MC-A 4	9	79	9	Cash Desk (CD)
MC-A 10	10	21,5	4,5	ZipMe
MC-A 2	11	40,015	55,02	Abstrakte SPL Unary
MC-A 2	13	511,54	716,78	Abstrakte SPL Unary
MC-A 2	15	0,52	0,22	Abstrakte SPL Binary
MC-A 2	15	1534,07	797,67	Abstrakte SPL Unary
MC-A 11	16	0,53	1,06	Minepump
MC-A 2	17	3271,125	1089,69	Abstrakte SPL Unary
MC-A 4	18	177	10	Cash Desk (CD/CH)
MC-A 10	20	79,43	42,02	Elevator Java
MC-A 10	20	36,35	4,16	Elevator C
MC-A 4	27	278	11	Cash Desk (CD/CT)
MC-A 2	31	3,63	1,34	Abstrakte SPL Binary
MC-A 2	31	0,75	0,17	Abstrakte SPL Ternary
MC-A 1	40	48,29	4,29	E-Mail Client
MC-A 10	40	36,32	16,6	E-Mail Java
MC-A 10	40	219,37	169,99	E-Mail C
MC-A 10	42	39,6	2,52	GPL
MC-A 4	54	652	12	Cash Desk (CD/CT/CH)
MC-A 2	63	32,035	20,04	Abstrakte SPL Binary
MC-A 3	64	40,36	13,08	Minepump
MC-A 10	64	177,5	8,34	Mine Pump Java
MC-A 10	64	104,12	3,57	Mine Pump C
MC-A 12	64	43,26	21,84	Minepump
MC-A 11	72	5,2	2,44	Minepump
MC-A 17	118	42,08	62,88	Coffee Machine (ohne Bisim.)
MC-A 17	118	38,53	17,44	Coffee Machine (mit Bisim.)
MC-A 2	127	223,46	147,81	Abstrakte SPL Binary
MC-A 2	127	14,585	2,28	Abstrakte SPL Ternary
MC-A 8	128	6,01	1,82	Minepump (SNIP –SPIN)
MC-A 20	186	271,68	24,74	Body Sensor Network (FABS)
MC-A 10	200	214	29,6	AJStats
MC-A 2	255	1306,495	1029,81	Abstrakte SPL Binary
MC-A 2	511	10573,015	8929,49	Abstrakte SPL Binary
MC-A 2	511	364,34	77,34	Abstrakte SPL Ternary
MC-A 7	512	75,82	38,18	Elevator
MC-A 11	512	14,87	2,72	Minepump
MC-A 2	2047	7073,04	1321,54	Abstrakte SPL Ternary
MC-A 21	17544	186,72	660	E-Server (MTBDD Engine)

Tabelle A.8.: Tabelle für die Hypothese GH1 für die Verifikationszeit für viele Produkte

Dokument	#Produkte	Zeit(s) PROD	Zeit(s) FAM	SPL
MC-A 2	3	0,025	0,01	Abstrakte SPL Binary
MC-A 12	4	9,3	10,09	Minepump
MC-A 2	5	0,075	0,05	Abstrakte SPL Unary
MC-A 2	7	0,095	0,04	Abstrakte SPL Binary
MC-A 2	7	0,06	0,02	Abstrakte SPL Ternary
MC-A 2	7	0,395	0,35	Abstrakte SPL Unary

Tabelle A.9.: Tabelle für Hypothese GH2 für die Verifikationszeit für wenige Produkte

Dokument	#Features	Zeit(s) PROD	Zeit(s) FAM	SPL
MC-A 2	7	0,52	0,22	Abstrakte SPL Binary
MC-A 2	7	0,75	0,17	Abstrakte SPL Ternary
MC-A 2	7	0,395	0,35	Abstrakte SPL Unary
MC-A 4	7	79	9	Cash Desk (CD)
MC-A 10	7	177,5	8,34	Mine Pump Java
MC-A 10	7	104,12	3,57	Mine Pump C
MC-A 10	8	21,5	4,5	ZipMe
MC-A 22	8	27,928	1,304	Failure Recovery-System
MC-A 22	8	29,224	0,52	Service Provider-System
MC-A 2	9	3,63	1,34	Abstrakte SPL Binary
MC-A 2	9	3,07	3,04	Abstrakte SPL Unary
MC-A 3	9	40,36	13,08	Minepump
MC-A 4	9	177	10	Cash Desk (CD/CH)
MC-A 7	9	75,82	38,18	Elevator
MC-A 10	9	36,32	16,6	E-Mail Java
MC-A 10	9	219,37	169,99	E-Mail C
MC-A 12	9	43,26	21,84	Minepump
MC-A 1	10	48,29	4,29	E-Mail Client
MC-A 2	10	14,585	2,28	Abstrakte SPL Ternary
MC-A 11	10	5,2	2,44	Minepump
MC-A 22	10	57,175	11,645	Failure Recovery-System
MC-A 22	10	57,215	0,68	Service Provider-System
MC-A 2	11	32,035	20,04	Abstrakte SPL Binary
MC-A 2	11	40,015	55,02	Abstrakte SPL Unary
MC-A 8	11	6,01	1,82	Minepump (SNIP –SPIN)
MC-A 20	11	271,68	24,74	Body Sensor Network (FABS)
MC-A 22	12	115,027	21,81	Failure Recovery-System
MC-A 22	12	119,544	1,64	Service Provider-System
MC-A 2	13	223,46	147,81	Abstrakte SPL Binary
MC-A 2	13	364,34	77,34	Abstrakte SPL Ternary
MC-A 2	13	511,54	716,78	Abstrakte SPL Unary
MC-A 11	13	14,87	2,72	Minepump
MC-A 21	14	186,72	660	E-Server (MTBDD Engine)

MC-A 22	14	226,233	44,942	Failure Recovery-System
MC-A 22	14	227,91	1,93	Service Provider-System
MC-A 2	15	1306,495	1029,81	Abstrakte SPL Binary
MC-A 2	15	1534,07	797,67	Abstrakte SPL Unary
MC-A 4	15	278	11	Cash Desk (CD/CT)
MC-A 2	16	7073,04	1321,54	Abstrakte SPL Ternary
MC-A 22	16	452,556	104,915	Failure Recovery-System
MC-A 22	16	466,185	2,97	Service Provider-System
MC-A 2	17	10573,015	8929,49	Abstrakte SPL Binary
MC-A 2	17	3271,125	1089,69	Abstrakte SPL Unary
MC-A 4	17	652	12	Cash Desk (CD/CT/CH)
MC-A 10	18	39,6	2,52	GPL
MC-A 17	18	42,08	62,88	Coffee Machine (ohne Bisim.)
MC-A 17	18	38,53	17,44	Coffee Machine (mit Bisim.)
MC-A 10	20	214	29,6	AJStats

Tabelle A.10.: Tabelle für die Hypothese GH3 für die Verifikationszeit für viele Features

Dokument	#Features	Zeit(s) PROD	Zeit(s) FAM	SPL
MC-A 22	2	3,4	0,044	Failure Recovery-System
MC-A 22	2	4,207	0,11	Service Provider-System
MC-A 2	3	0,025	0,01	Abstrakte SPL Binary
MC-A 2	4	0,06	0,02	Abstrakte SPL Ternary
MC-A 12	4	9,3	10,09	Minepump
MC-A 22	4	7,105	0,078	Failure Recovery-System
MC-A 22	4	6,932	0,31	Service Provider-System
MC-A 2	5	0,095	0,04	Abstrakte SPL Binary
MC-A 2	5	0,075	0,05	Abstrakte SPL Unary
MC-A 10	6	79,43	42,02	Elevator Java
MC-A 10	6	36,35	4,16	Elevator C
MC-A 11	6	0,53	1,06	Minepump
MC-A 22	6	14,564	0,349	Failure Recovery-System
MC-A 22	6	15,57	0,31	Service Provider-System

Tabelle A.11.: Tabelle für die Hypothese GH4 für die Verifikationszeit für wenige Features

Dokument	LoC	Zeit(s) PROD	Zeit(s) FAM	SPL
MC-A 10	1233	36,32	16,6	E-Mail Java
MC-A 10	258	219,37	169,99	E-Mail C
MC-A 10	1046	79,43	42,02	Elevator Java
MC-A 10	877	36,35	4,16	Elevator C
MC-A 10	580	177,5	8,34	Mine Pump Java
MC-A 10	279	104,12	3,57	Mine Pump C
MC-A 10	13393	214	29,6	AJStats
MC-A 10	1405	39,6	2,52	GPL
MC-A 10	3636	21,5	4,5	ZipMe

Tabelle A.12.: Tabelle für die Hypothese GH5 für die Verifikationszeit in Abhängigkeit der Anzahl Codezeilen

Dokument	#Zustände $PROD_{PARA}$	#Zustände FAM	SPL
MC-A 5	854	774	Coffee Machine
MC-A 5	15533	14342	Coffee Machine
MC-A 5	44548	26179	Coffee Machine
MC-A 5	79217	38016	Coffee Machine
MC-A 5	191493	96682	Coffee Machine
MC-A 5	447203	245006	Coffee Machine
MC-A 8	613179	180857	Minepump (mit SNIP -SPIN)
MC-A 13	637314	591040	Cash Desk
MC-A 13	725517	685230	Elevator
MC-A 13	15374	2712	Vending Machine
MC-A 13	15191	5697	Minepump

Tabelle A.13.: Tabelle für die Hypothese GH6 bezüglich Speicherverbrauch von parallelem Produkt-basierten Model Checking und Familien-basierten Model Checking

Dokument	#Zustände $PROD_{SEQ}$	#Zustände FAM	SPL
MC-A 5	427	774	Coffee Machine
MC-A 5	3884	14342	Coffee Machine
MC-A 5	5569	26179	Coffee Machine
MC-A 5	6602	38016	Coffee Machine
MC-A 5	7979	96682	Coffee Machine
MC-A 5	12423	245006	Coffee Machine
MC-A 8	4791	180857	Minepump (mit SNIP -SNIP)
MC-A 13	70813	591040	Cash Desk
MC-A 13	181380	685230	Elevator
MC-A 13	615	2712	Vending Machine
MC-A 13	3798	5697	Minepump

Tabelle A.14.: Tabelle für die Hypothese GH7 bezüglich Speicherverbrauch von sequenziellen Produkt-basierten Model Checking und Familien-basierten Model Checking

Hypothese	R-Befehl
GH1	*wilcox.test(zeitFam,zeitProd,paired=TRUE,alternative="less")*
GH2	*wilcox.test(zeitProd,zeitFam,paired=TRUE,alternative="less")*
GH3	*wilcox.test(zeitFam,zeitProd,paired=TRUE,alternative="less")*
GH4	*wilcox.test(zeitProd,zeitFam,paired=TRUE,alternative="less")*
GH5	*wilcox.test(zeitFam,zeitProd,paired=TRUE,alternative="less")*
GH6	*wilcox.test(zustFam,zustProdPara,paired=TRUE,alternative="less")*
GH7	*wilcox.test(zustProdSeq,zustFam,paired=TRUE,alternative="less")*

Tabelle A.16.: Auflistung der R-Befehle zur Ausführung der Wilcoxon-Vorzeichen-Rang-Tests für die jeweilige Hypothese bezüglich Model Checking.

A.2.2. Tabellen zum Type Checking

Dieser Abschnitt enthält die Tabellen, welche die Datensätze zu den Diagrammen und Untersuchungen aus Abschnitt 4.3 darstellen. Die Tabellen A.18 und A.20 stellen die berechneten Standardabweichungen und Varianzen sowie die daraus ermittelten Effektgrößen d_{Cohen} der entsprechenden Primärstudien dar. Daraufhin werden in den Tabellen A.21 bis A.25 die jeweiligen unabhängigen und abhängigen Variablen der jeweiligen untersuchten Hypothese und die entsprechende abhängige Variable aufgelistet. Diese Daten entsprechen ebenfalls den mit den Meta-Diagrammen des Type Checkings visualisierten Daten.

Dok.	Mittelwert PROD	Standard-abweichung PROD	Mittelwert FAM	Standard-abweichung FAM	d_{Cohen}	n	95% Konfidenzintervall
TC-A 1	157	376	2,27	2,65	-0,583	12	[-1,74 ; 0,573]
TC-A 11	1872001238	4185918562	10.3	9.32	-0,632	5	[-2,43 ; 1,16]
TC-A 13	89,1	223	1,56	1,74	-0,554	12	[-1,71 ; 0,599]
TC-A 14	375	474	97,1	122	-0,803	7	[-2,34 ; 0.737]
TC-A 16	623	1448	1172	2116	0,303	11	[-0,886 ; 1,49]
TC-A 18	1147667	1376661	0.789	0.318	-1,18	3	[-3,63 ; -1,27]

Tabelle A.18.: Mittelwerte und Standardabweichungen der Produkt-basierten und Familien-basierten Strategie der einzelnen Type-Checking-Primärstudien , berechneter d_{Cohen}-Wert inklusive den 95 % Konfidenzintervallen

Dok.	Strategie A	Strategie B	Mittelwert Strategie A	Standard-abweichung Strategie A	Mittelwert Strategie B	Standard-abweichung Strategie B	d_{Cohen}	n	95% Konfidenzintervall
TC-A 1	PROD	FEAT	310	728	3,86	4,06	-0,577	12	[-1,73 ; 0,578]
TC-A 1	PROD	FEAT-PROD	310	728	255	611	-0,082	12	[-1,215 ; 1,05]
TC-A 13	PROD	FEAT-PROD	89,1	223	198	500	0,281	12	[-0,856 ; 1,42]
TC-A 1	FEAT	FAM	3,86	4,06	2,27	2,65	-0,464	12	[-1,61 ; 0,682]
TC-A 1	FEAT-PROD	FAM	255	611	2,88	2,77	-0,584	12	[-1,74 ; 0.571]
TC-A 13	FEAT-PROD	FAM	198	500	1,56	1,74	-0,556	12	[-1,71 ; 0.598]
TC-A 1	FEAT-PROD	FEAT	255	611	12,2	14,4	-0,562	12	[-1,72 ; 0,591]

Tabelle A.20.: Mittelwerte und Standardabweichungen der weiteren miteinander verglichenen Strategien der einzelnen Type-Checking-Primärstudien , berechneter d_{Cohen}-Wert inklusive den 95 % Konfidenzintervallen

Dokument	#Produkte	Zeit(s) PROD	Zeit(s) FAM	SPL
TC-A 1	16	2,8	0,3	GraphLib
TC-A 1	16	3,1	0,3	Raroscope
TC-A 5	16	334,7	34,3	NO SPL
TC-A 5	16	380,5	55,0	NO SPL
TC-A 13	16	1,17	0,18	Graph
TC-A 13	16	1,35	0,18	Raroscope
TC-A 16	16	13,99	3,08	Graph
TC-A 16	16	15,84	2,62	Raroscope
TC-A 1	24	22,5	3,7	GuiDSL
TC-A 1	24	10,2	1	ZipMe
TC-A 13	24	14,77	2,67	GuiDSL
TC-A 13	24	7,86	0,73	ZipMe
TC-A 16	24	90,54	4154,96	GuiDSL
TC-A 16	24	47,04	379,94	ZipMe
TC-A 1	32	35,7	2,2	Prevayler
TC-A 13	32	21,66	1,43	Prevayler
TC-A 16	32	108,51	52,63	Prevayler
TC-A 18	32	82000	0,55	ZipMe
TC-A 1	40	37,1	10	Violet
TC-A 13	40	20,25	6,55	Violet
TC-A 1	48	41,6	2	PKJab
TC-A 13	48	23,33	1,60	PKJab
TC-A 16	48	137,1	112,17	PKJab
TC-A 1	64	37,9	1,1	Sudoku
TC-A 13	64	17,71	0,73	Sudoku
TC-A 16	64	126,07	20,24	Sudoku
TC-A 11	144	28,8	1,3	MobileMedia Release 6
TC-A 1	156	43,7	1,7	GPL
TC-A 13	156	19,07	1,15	GPL
TC-A 16	156	189,12	1823,38	GPL
TC-A 1	425	28,9	0,4	EPL
TC-A 13	425	16,64	0,28	EPL
TC-A 16	425	303,44	12,18	EPL
TC-A 1	512	300,1	1,6	Notepad
TC-A 13	512	135,52	1,20	Notepad
TC-A 16	512	903,86	81,01	Notepad
TC-A 18	840	2702000	1,15	GPL
TC-A 18	1024	659000	0,667	Notepad
TC-A 11	2048	1228,8	8,3	Mobile RSS Reader
TC-A 11	2048	4096	19,0	Lampiro
TC-A 1	2458	1326,5	2,9	TankWar
TC-A 13	2458	790,01	2,01	TankWar
TC-A 16	2458	4925,82	6258,69	TankWar

TC-A 11	2784	835,2	1,8	MobileMedia Release 8
TC-A 11	$3{,}6 * 10^9$	$9{,}36 * 10^9$	21,0	Berkeley DB
TC-A 7	$\sim 1{,}29 * 10^{33}$	$5{,}19 * 10^{33}$	60	Boa (Webserver)

Tabelle A.21.: Tabelle für die Hypothese GH1 für die Verifikationszeit für viele Produkte

Dokument	**#Produkte**	**Zeit(s) PROD**	**Zeit(s) FAM**	**SPL**
TC-A 5	4	10,2	10,1	NO SPL
TC-A 5	4	10,1	15,7	NO SPL
TC-A 5	4	125,1	63,0	NO SPL
TC-A 5	8	20,4	10,2	NO SPL

Tabelle A.22.: Tabelle für die Hypothese GH2 für die Verifikationszeit für wenige Produkte

Dokument	#Features	Zeit(s) PROD	Zeit(s) FAM	SPL
16)	7	126,07	20,24	Sudoku
TC-A 1	8	41,6	2	PKJab
TC-A 13	8	23,33	1,60	PKJab
TC-A 13	8	17,71	0,73	Sudoku
TC-A 16	8	137,1	112,17	PKJab
TC-A 11	9	28,8	1,3	MobileMedia Release 6
TC-A 1	10	300,1	1,6	Notepad
TC-A 11	11	4096	19,0	Lampiro
TC-A 1	12	28,9	0,4	EPL
TC-A 13	12	16,64	0,28	EPL
TC-A 16	12	303,44	12,18	EPL
TC-A 16	12	903,86	81,01	Notepad
TC-A 1	13	10,2	1	ZipMe
TC-A 13	13	135,52	1,20	Notepad
TC-A 13	13	7,86	0,73	ZipMe
TC-A 16	13	47,04	379,94	ZipMe
TC-A 11	14	835,2	1,8	MobileMedia Release 8
TC-A 11	14	1228,8	8,3	Mobile RSS Reader
TC-A 18	16	659000	0,667	Notepad
TC-A 18	17	82000	0,55	ZipMe
TC-A 1	25	43,7	1,7	GPL
TC-A 1	26	22,5	3,7	GuiDSL
TC-A 16	26	90,54	4154,96	GuiDSL
TC-A 13	27	19,07	1,15	GPL
TC-A 16	27	189,12	1823,38	GPL
TC-A 13	28	14,77	2,67	GuiDSL
TC-A 13	37	790,01	2,01	TankWar
TC-A 16	37	4925,82	6258,69	TankWar
TC-A 1	38	1326,5	2,9	TankWar
TC-A 18	38	2702000	1,15	GPL
TC-A 11	42	$9{,}36 * 10^{9}$	21,0	Berkeley DB
TC-A 1	88	37,1	10	Violet
TC-A 13	89	20,25	6,55	Violet
TC-A 7	110	$5{,}19 * 10^{33}$	60	Boa (Webserver)

Tabelle A.23.: Tabelle für die Hypothese GH3 für die Verifikationszeit für viele Features

Dokument	#Features	Zeit(s) PROD	Zeit(s) FAM	SPL
TC-A 1	4	3,1	0,3	Raroscope
TC-A 1	5	2,8	0,3	GraphLib
TC-A 13	5	1,17	0,18	Graph
TC-A 13	5	1,35	0,18	Raroscope
TC-A 16	5	13,99	3,08	Graph

TC-A 16	5	15,84	2,62	Raroscope
TC-A 1	6	35,7	2,2	Prevayler
TC-A 1	6	37,9	1,1	Sudoku
TC-A 13	6	21,66	1,43	Prevayler
TC-A 16	6	108,51	52,63	Prevayler

Tabelle A.24.: Tabelle für die Hypothese GH4 für die Verifikationszeit für wenige Features

Dokument	LoC	Zeit(s) PROD	Zeit(s) FAM	SPL
TC-A 13	2130	17,71	0,73	Sudoku
TC-A 1	2193	300,1	1,6	Notepad
TC-A 18	2727	659000	0,667	Notepad
TC-A 13	2732	135,52	1,20	Notepad
TC-A 1	2855	43,7	1,7	GPL
TC-A 13	2951	19,07	1,15	GPL
TC-A 18	2955	2702000	1,15	GPL
TC-A 16	3373	137,1	112,17	PKJab
TC-A 16	3446	47,04	379,94	ZipMe
TC-A 1	4109	41,6	2	PKJab
TC-A 11	4600	28,8	1,3	MobileMedia Release 6
TC-A 1	4845	1326,5	2,9	TankWar
TC-A 16	4845	4925,82	6258,69	TankWar
TC-A 18	4988	82000	0,55	ZipMe
TC-A 13	5000	23,33	1,60	PKJab
TC-A 13	5069	7,86	0,73	ZipMe
TC-A 1	5076	10,2	1	ZipMe
TC-A 16	5268	108,51	52,63	Prevayler
TC-A 13	5604	790,01	2,01	TankWar
TC-A 11	5700	835,2	1,8	MobileMedia Release 8
TC-A 13	5938	21,66	1,43	Prevayler
TC-A 1	6185	35,7	2,2	Prevayler
TC-A 7	6200	$5{,}19 * 10^{33}$	60	Boa (Webserver)
TC-A 16	10084	90,54	4154,96	GuiDSL
TC-A 1	10866	37,1	10	Violet
TC-A 13	11006	20,25	6,55	Violet
TC-A 1	14318	22,5	3,7	GuiDSL
TC-A 13	15988	14,77	2,67	GuiDSL
TC-A 11	20000	1228,8	8,3	Mobile RSS Reader
TC-A 11	45000	4096	19,0	Lampiro
TC-A 11	70000	$9{,}36 * 10^{9}$	21,0	Berkeley DB

Tabelle A.25.: Tabelle für die Hypothese GH5 für die Verifikationszeit in Abhängigkeit der Anzahl Codezeilen

A.2.3. Tabellen für die Untersuchung der Performanz bezüglich Static Analysis

In den folgenden Tabellen A.26 und A.27 sind die Daten der Meta-Diagramme bezüglich Static Analysis dargestellt.

Dok.	#F	#P	LoC	Zeit(s) PROD	SPL
SA-A 1	18	106	1350	0,524235	Graph PL - A0 inkl. Generierungszeit
SA-A 1	14	24	5700	0,514528	MobileMedia Rel. 8 - A0 inkl. Generierungszeit
SA-A 1	5	8	8000	0,164078	Prevayler - A0 inkl. Generierungszeit
SA-A 1	11	4	45000	2,678049	Lampiro - A0 inkl. Generierungszeit
SA-A 1	42	40	84000	5,63166	BerkeleyDB - A0 inkl. Generierungszeit
SA-A 7	NA	106	NA	0,1793	GPL::V'.display() - LIFTED+SHARING
SA-A 7	NA	72	NA	0,00436	GPL::G'.run() - LIFTED+SHARING
SA-A 7	NA	40	NA	0,19783	BerkeleyDB::D'.main() - LIFTED+SHARING
SA-A 7	NA	8	NA	0,00035	Prevayler::P'.publisher() - LIFTED+SHARING
SA-A 7	NA	106	NA	0,1793	GPL::V'.display() - LIFTED+MAX ABSTRACTI-ON
SA-A 7	NA	72	NA	0,00436	GPL::G'.run() - LIFTED+MAX ABSTRACTION
SA-A 7	NA	40	NA	0,19783	BerkeleyDB::D'.main() - LIFTED+MAX AB-STRACTION
SA-A 7	NA	8	NA	0,00035	Prevayler::P'.publisher() - LIFTED+MAX AB-STRACTION

Tabelle A.26.: Tabelle der Verifikationszeit der Produkt-basierten Strategie für die Meta-Diagramme aus Kapitel 4.4

Dok.	#F	#P	LoC	Zeit(s) FAM	SPL
SA-A 1	18	106	1350	0,129	Graph PL - A1 inkl. Generierungszeit
SA-A 1	18	106	1350	0,074	Graph PL - A2 inkl. Generierungszeit
SA-A 1	18	106	1350	0,0676	Graph PL - A3 inkl. Generierungszeit
SA-A 1	14	24	5700	0,184	MobileMedia Rel. 8 - A1 inkl. Generierungszeit
SA-A 1	14	24	5700	0,149	MobileMedia Rel. 8 - A2 inkl. Generierungszeit
SA-A 1	14	24	5700	0,149	MobileMedia Rel. 8 - A3 inkl. Generierungszeit
SA-A 1	5	8	8000	0,140	Prevayler - A1 inkl. Generierungszeit
SA-A 1	5	8	8000	0,137	Prevayler - A2 inkl. Generierungszeit
SA-A 1	5	8	8000	0,139	Prevayler - A3 inkl. Generierungszeit
SA-A 1	11	4	45000	2,766	Lampiro - A1 inkl. Generierungszeit
SA-A 1	11	4	45000	2,861	Lampiro - A2 inkl. Generierungszeit
SA-A 1	11	4	45000	2,824	Lampiro - A3 inkl. Generierungszeit
SA-A 1	42	40	84000	2,216	BerkeleyDB - A1 inkl. Generierungszeit
SA-A 1	42	40	84000	2,045	BerkeleyDB - A2 inkl. Generierungszeit
SA-A 1	42	40	84000	1,934	BerkeleyDB - A3 inkl. Generierungszeit
SA-A 2	56	$55 * 10^{10}$	84000	24	BerkeleyDB - Possible Types
SA-A 2	29	524288	1400	42	GPL - Possible Types

SA-A 2	20	4	45000	4	Lampiro - Possible Types
SA-A 2	34	512	5700	3	MobileMedia Release 8 - Possible Types
SA-A 2	56	$55 * 10^{10}$	84000	724	BerkeleyDB - Reaching Definitions
SA-A 2	29	524288	1400	528	GPL - Reaching Definitions
SA-A 2	20	4	45000	42	Lampiro - Reaching Definitions
SA-A 2	34	512	5700	59	MobileMedia Release 8 - Reaching Definitions
SA-A 2	56	$55 * 10^{10}$	84000	618	BerkeleyDB - Uninitialized Variables
SA-A 2	29	524288	1400	429	GPL - Uninitialized Variables
SA-A 2	20	4	45000	85	Lampiro - Uninitialized Variables
SA-A 2	34	512	5700	133	MobileMedia Release 8 - Uninitialized Variables
SA-A 3	14	NA	257	22,172	Scari (DirectSE)
SA-A 3	14	NA	257	9,814	Scari (IncComp)
SA-A 3	13	NA	590	981,67	GPL (DirectSE)
SA-A 3	13	NA	590	272,52	GPL (IncComp)
SA-A 4	18	156	1713	72	GPL (CycleTest)
SA-A 4	18	156	1713	72	GPL (StronglyConnectedTest)
SA-A 4	25	7057	2074	2856	Notepad (PersistenceTest)
SA-A 4	25	7057	2074	2671	Notepad (PrintTest)
SA-A 4	17	5	26332	254	jak2java (ReduceToJavaTest)
SA-A 4	17	5	26332	100	jak2java (ArgInquireTest)
SA-A 5	18	156	1713	69,4	GPL (DisplayCheck)
SA-A 5	18	156	1713	110,2	GPL (SearchCheck)
SA-A 5	18	156	1713	69,8	GPL (KruskalCheck)
SA-A 5	25	7057	2074	296,3	Notepad (PersistenceCheck)
SA-A 5	25	7057	2074	259,9	Notepad (CopyPasteCheck)
SA-A 5	25	7057	2074	279,8	Notepad (UndoRedoCheck)
SA-A 6	5	8	NA	1	WS original (ohne Abstraktion)
SA-A 6	6	12	NA	1	WS evolved (ohne Abstraktion)
SA-A 6	8	42	NA	2	VM original (ohne Abstraktion)
SA-A 6	8	42	NA	2	VM evolved (ohne Abstraktion)
SA-A 6	5	18	NA	78,5	CD original (ohne Abstraktion)
SA-A 6	5	18	NA	70	CD evolved (ohne Abstraktion)
SA-A 6	6	8	NA	279	MP original (ohne Abstraktion)
SA-A 6	7	16	NA	763,5	MP evolved (ohne Abstraktion)
SA-A 6	3	8	NA	254	ES original (ohne Abstraktion)
SA-A 6	2	4	NA	0	ES evolved (ohne Abstraktion)
SA-A 6	5	8	NA	1	WS original (mit Abstraktion)
SA-A 6	6	12	NA	0,5	WS evolved (mit Abstraktion)
SA-A 6	8	42	NA	1	VM original (mit Abstraktion)
SA-A 6	8	42	NA	0,5	VM evolved (mit Abstraktion)
SA-A 6	5	18	NA	18	CD original (mit Abstraktion)
SA-A 6	5	18	NA	7,5	CD evolved (mit Abstraktion)

SA-A 6	6	8	NA	168	MP original (mit Abstraktion)
SA-A 6	7	16	NA	535	MP evolved (mit Abstraktion)
SA-A 6	3	8	NA	153,5	ES original (mit Abstraktion)
SA-A 7	NA	106	NA	0,0231	GPL::V'.display() - LIFTED+SHARING
SA-A 7	NA	72	NA	$8{,}7 * 10^{-4}$	GPL::G'.run() - LIFTED+SHARING
SA-A 7	NA	40	NA	$8{,}32 * 10^{-2}$	BerkeleyDB::D'.main() - LIFTED+SHARING
SA-A 7	NA	8	NA	$1{,}8 * 10^{-4}$	Prevayler::P'.publisher() - LIFTED+SHARING
SA-A 7	NA	106	NA	$1{,}17 * 10^{-3}$	GPL::V'.display() - LIFTED+MAX ABSTRACTION
SA-A 7	NA	72	NA	$1{,}1 * 10^{-3}$	GPL::G'.run() - LIFTED+MAX ABSTRACTION
SA-A 7	NA	40	NA	$5{,}05 * 10^{-3}$	BerkeleyDB::D'.main() - LIFTED+MAX ABSTRACTION
SA-A 7	NA	8	NA	$6{,}5 * 10^{-5}$	Prevayler::P'.publisher() - LIFTED+MAX ABSTRACTION
SA-A 8	5	NA	8000	$1{,}92 * 10^{-4}$	P'F'.publisher() - Reaching Definitions - Baseline A2
SA-A 8	5	NA	8000	$1{,}84 * 10^{-4}$	P'F'.publisher() - Reaching Definitions - Sharing
SA-A 8	5	NA	8000	$1{,}28 * 10^{-4}$	P'F'.publisher() - Reaching Definitions - Med. Abstraction
SA-A 8	5	NA	8000	$6{,}75 * 10^{-5}$	P'F'.publisher() - Reaching Definitions - Max. Abstraction
SA-A 8	42	NA	84000	0,125	DBRunAction.main() - Reaching Definitions - Baseline A2
SA-A 8	42	NA	84000	0,107	DBRunAction.main() - Reaching Definitions - Sharing
SA-A 8	42	NA	84000	0,0614	DBRunAction.main() - Reaching Definitions - Med. Abstraction
SA-A 8	42	NA	84000	$1{,}76 * 10^{-3}$	DBRunAction.main() - Reaching Definitions - Max. Abstraction
SA-A 8	18	NA	1350	0,0455	Vertex.display() - Reaching Definitions - Baseline A2
SA-A 8	18	NA	1350	0,025	Vertex.display() - Reaching Definitions - Sharing
SA-A 8	18	NA	1350	0,0269	Vertex.display() - Reaching Definitions - Med. Abstraction
SA-A 8	18	NA	1350	$3{,}7 * 10^{-4}$	Vertex.display() - Reaching Definitions - Max. Abstraction
SA-A 8	5	NA	8000	$2{,}61 * 10^{-4}$	P'F'.publisher() - Uninitialized Variables - Baseline A2
SA-A 8	5	NA	8000	$2{,}1 * 10^{-4}$	P'F'.publisher() - Uninitialized Variables - Sharing
SA-A 8	5	NA	8000	$1{,}66 * 10^{-4}$	P'F'.publisher() - Uninitialized Variables - Med. Abstraction
SA-A 8	5	NA	8000	$8{,}91 * 10^{-5}$	P'F'.publisher() - Uninitialized Variables - Max. Abstraction
SA-A 8	42	NA	84000	0,0189	DBRunAction.main() - Uninitialized Variables - Baseline A2

SA-A 8	42	NA	84000	0,0155	DBRunAction.main() - Uninitialized Variables - Sharing
SA-A 8	42	NA	84000	$9{,}51 * 10^{-3}$	DBRunAction.main() - Uninitialized Variables - Med. Abstraction
SA-A 8	42	NA	84000	$1{,}31 * 10^{-3}$	DBRunAction.main() - Uninitialized Variables - Max. Abstraction
SA-A 8	18	NA	1350	0,0178	Vertex.display() - Uninitialized Variables - Baseline A2
SA-A 8	18	NA	1350	$8{,}23 * 10^{-3}$	Vertex.display() - Uninitialized Variables - Sharing
SA-A 8	18	NA	1350	$9{,}31 * 10^{-3}$	Vertex.display() - Uninitialized Variables - Med. Abstraction
SA-A 8	18	NA	1350	$5{,}42 * 10^{-4}$	Vertex.display() - Uninitialized Variables - Max. Abstraction
SA-A 9	792	NA	191615	310	Busybox
SA-A 9	6918	NA	$6{,}5 * 10^5$	26200	Linux
SA-A 9	589	NA	233450	380	OpenSSL
SA-A 10	792	NA	191615	20,1	Busybox
SA-A 10	6918	NA	$6{,}7 * 10^5$	2610	Linux
SA-A 11	792	NA	191615	97,3	Busybox
SA-A 11	6918	NA	$6{,}5 * 10^5$	13900	Linux
SA-A 11	589	NA	233450	44,7	OpenSSL

Tabelle A.27.: Tabelle der Verifikationszeit der Familien-basierten Strategie für die Meta-Diagramme aus Kapitel 4.4

A.2.4. Tabellen für die Untersuchung der Performanz bezüglich Theorem Proving

Die folgenden zwei Tabellen A.28 und A.29 enthalten die Daten der Meta-Diagramme des Theorem Provings aus Abschnitt 4.5.

Dokument	#Features	#Produkte	LoC	Zeit(s) PROD	SPL
TP-A 1	2	2	75	3,152	BankAccount
TP-A 1	3	4	86	7,03	BankAccount
TP-A 1	4	8	124	22,545	BankAccount
TP-A 1	5	12	135	39,272	BankAccount
TP-A 1	6	24	140	83,636	BankAccount
TP-A 5	1	1	51	7,62	BankAccount
TP-A 5	2	2	57	15,24	BankAccount
TP-A 5	3	4	95	35,58	BankAccount
TP-A 5	4	6	106	53,34	BankAccount

TP-A 5	5	12	117	114,3	BankAccount
TP-A 5	6	24	172	261,6	BankAccount
TP-A 5	7	48	198	561,24	BankAccount
TP-A 5	8	72	239	916,8	BankAccount

Tabelle A.28.: Tabelle der Verifikationszeit der Produkt-basierten Strategie für die Meta-Diagramme aus Kapitel 4.5

Dokument	#Features	#Produkte	LoC	Zeit(s) FAM	SPL
TP-A 1	2	2	75	3,152	BankAccount
TP-A 1	3	4	86	3,394	BankAccount
TP-A 1	4	8	124	8,242	BankAccount
TP-A 1	5	12	135	8,242	BankAccount
TP-A 1	6	24	140	12,364	BankAccount
TP-A 4	8	72	239	34,778	BankAccount
TP-A 4	8	72	239	149,216	BankAccount
TP-A 5	1	1	51	7,62	BankAccount
TP-A 5	2	2	57	10,14	BankAccount
TP-A 5	3	4	95	25,38	BankAccount
TP-A 5	4	6	106	25,38	BankAccount
TP-A 5	5	12	117	27,96	BankAccount
TP-A 5	6	24	172	38,1	BankAccount
TP-A 5	7	48	198	43,2	BankAccount
TP-A 5	8	72	239	114,3	BankAccount
TP-A 6	1	1	NA	1,47	BankAccount
TP-A 6	2	2	NA	3,98	BankAccount
TP-A 6	3	4	NA	6,56	BankAccount
TP-A 6	4	6	NA	7,12	BankAccount
TP-A 6	5	12	NA	6,89	BankAccount
TP-A 6	6	24	NA	7,19	BankAccount
TP-A 6	7	36	NA	8,12	BankAccount
TP-A 6	8	36	NA	19,48	BankAccount
TP-A 6	9	72	NA	123,8	BankAccount
TP-A 6	10	144	NA	371,81	BankAccount

Tabelle A.29.: Tabelle der Verifikationszeit der Familien-basierten Strategie für die Meta-Diagramme aus Kapitel 4.5

A.3. Quellcode des SPL für Mp3-Player

Listing A.1: Grafische Schnittstelle des Mp3-Players

```
1  package Framework;
2
3  import java.awt.Color;
4  import java.awt.FlowLayout;
5  import java.awt.GridLayout;
6  import java.awt.event.ActionEvent;
7  import java.awt.event.ActionListener;
8  import java.io.IOException;
9
10 import javax.sound.sampled.LineUnavailableException;
11 import javax.sound.sampled.UnsupportedAudioFileException;
12 import javax.swing.JButton;
13 import javax.swing.JFileChooser;
14 import javax.swing.JFrame;
15 import javax.swing.JLabel;
16 import javax.swing.JMenu;
17 import javax.swing.JMenuBar;
18 import javax.swing.JMenuItem;
19 import javax.swing.JPanel;
20 import javax.swing.JTextArea;
21 import javax.swing.Timer;
22 import javax.swing.filechooser.FileNameExtensionFilter;
23
24 import Interfaces.AdditionalButtonIF;
25 import Interfaces.ProgressIF;
26
27 import com.mpatric.mp3agic.InvalidDataException;
28 import com.mpatric.mp3agic.UnsupportedTagException;
29
30
31
32 //import pluginInterfaces.ClearPlaylistPluginIF;
33
34 /**
35  *
36  * @author Vasco A simple graphical user interface for the MP3Player which
37  *         enables the user to choose a mp3 file to be played and enables to
38  *         start, pause and stop the song.
39  *
40  */
41
42 public class AMPGui {
```

```
43
44    private JFrame amp;
45    private JPanel container;
46    private JPanel mp3Titles;
47    private JPanel playEnginePanel;
48    private JPanel progressPanel;
49    private JPanel additionalButtonPanel;
50
51    private JButton play;
52    private JButton pause;
53    private JButton stop;
54    private JMenuBar menuBar;
55    private JMenu file;
56    private JMenuItem lookUpFile;
57    ActionListener playedTimeListener;
58    Timer playedTimer;
59
60    MP3Player mp3Player;
61    JFileChooser fc;
62
63    JLabel songInfo;
64
65    JLabel progress;
66    ProgressIF progressPlugin = null;
67    AdditionalButtonIF additionalButtonPlugin = null;
68
69    //TitleBar
70    private JTextArea information;
71
72    public AMPGui() {
73
74    }
75
76    public AMPGui(ProgressIF progressPlugin) {
77      this.progressPlugin = progressPlugin;
78      progressPlugin.registerGui(this);
79    }
80
81    public AMPGui(ProgressIF progressPlugin, AdditionalButtonIF
        additionalButtonPlugin) {
82      this.progressPlugin = progressPlugin;
83      progressPlugin.registerGui(this);
84      this.additionalButtonPlugin = additionalButtonPlugin;
85      additionalButtonPlugin.registerGui(this);
86    }
87
```

```
88  /**
89   * Initializes the GUI
90   */
91  public void createAndShowGui() {
92    amp = new JFrame("MP3-Player");
93    amp.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
94    amp.setLayout(new GridLayout(1, 2));
95    amp.setSize(250, 250);
96    menuBar = new JMenuBar();
97    file = new JMenu("Datei");
98    menuBar.add(file);
99    fc = new JFileChooser();
100
101   //Select Mp3 File
102   lookUpFile = new JMenuItem("Datei oeffnen");
103   lookUpFile.addActionListener(new ActionListener() {
104     @Override
105     public void actionPerformed(ActionEvent arg0) {
106       // The following lines restrict the file chooser for mp3 files
107       fc.setAcceptAllFileFilterUsed(false);
108       FileNameExtensionFilter filter = new FileNameExtensionFilter(
109           "MP3 songs", "mp3");
110       fc.addChoosableFileFilter(filter);
111       int returnVal = fc.showOpenDialog(amp);
112       if (returnVal == JFileChooser.APPROVE_OPTION) {
113         String filePath = fc.getSelectedFile().getAbsolutePath();
114         try {
115           if (getMp3Player() == null) {
116               setMp3Player(new MP3Player(filePath));
117           } else {
118               getMp3Player().close();
119               setMp3Player(null);
120               setMp3Player(new MP3Player(filePath));
121             }
122           boolean ready = false;
123           while (!ready) {
124             if (getMp3Player().isReady()) {
125               progress.setText(progressPlugin.refreshStatus());
126               progress.revalidate();
127               if (getMp3Player().getPlayerStatus() != 1) {
128                 getMp3Player().play();
129               }
130               getPlayedTimer().start();
131               getPlayedTimer().setDelay(1000);
132               ready = true;
133             }
```

```
134             }
135             songInfo.setText(mp3Player.getID3InformationWithoutProgress());
136           } catch (UnsupportedAudioFileException e) {
137             e.printStackTrace();
138           } catch (IOException e) {
139             e.printStackTrace();
140           } catch (LineUnavailableException e) {
141             e.printStackTrace();
142           } catch (UnsupportedTagException e) {
143             e.printStackTrace();
144           } catch (InvalidDataException e) {
145             e.printStackTrace();
146           }
147         }
148       }
149     });
150
151     file.add(lookUpFile);
152     amp.setJMenuBar(menuBar);
153
154     //Standard Layout Setting
155     container = new JPanel();
156     container.setLayout(new GridLayout(4, 1));
157     container.setBackground(new Color(236, 236, 236));
158     amp.getContentPane().add(container);
159     amp.setVisible(true);
160     amp.setResizable(true);
161
162     playEnginePanel = new JPanel();
163     playEnginePanel.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20));
164     playEnginePanel.setBackground(new Color(236, 236, 236));
165
166     //Start, Stop and Pause Button
167     play = new JButton("Start");
168     play.addActionListener(new ActionListener() {
169       @Override
170       public void actionPerformed(ActionEvent arg0) {
171         if (getMp3Player() != null && getMp3Player().isReady()) {
172           getMp3Player().play();
173           getPlayedTimer().start();
174         }
175       }
176     });
177
178     pause = new JButton("Pause");
179     pause.addActionListener(new ActionListener() {
```

```
180         @Override
181         public void actionPerformed(ActionEvent arg0) {
182           if (getMp3Player() != null) {
183             getMp3Player().pause();
184             if (getPlayedTimer().isRunning()) {
185               getPlayedTimer().stop();
186             }
187           }
188         }
189       });
190
191       stop = new JButton("Stop");
192       stop.addActionListener(new ActionListener() {
193         @Override
194         public void actionPerformed(ActionEvent arg0) {
195           if (getMp3Player() != null) {
196             getMp3Player().stop();
197           }
198         }
199       });
200
201       playEnginePanel.add(play);
202       playEnginePanel.add(pause);
203       playEnginePanel.add(stop);
204
205       //Listener for the song progress
206       ActionListener playedTimeListener =  new ActionListener() {
207         @Override
208         public void actionPerformed(ActionEvent arg0) {
209           if (getMp3Player().getPlayerStatus() == 1) {
210             progress.setText(progressPlugin.refreshStatus());
211             progress.revalidate();
212           }
213         }
214       };
215       playedTimer = new Timer(0, playedTimeListener);
216
217       container.add(playEnginePanel);
218
219       mp3Titles = new JPanel();
220       mp3Titles.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20));
221       mp3Titles.setBackground(new Color(236, 236, 236));
222
223       songInfo = new JLabel("No song loaded");
224       mp3Titles.add(songInfo);
225       container.add(mp3Titles);
```

```
226
227     progressPanel = new JPanel();
228     progressPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20));
229     progressPanel.setBackground(new Color(236, 236, 236));
230     progress = new JLabel();
231     progressPanel.add(progress);
232     container.add(progressPanel);
233
234     if(additionalButtonPlugin != null) {
235       additionalButtonPanel = new JPanel();
236       additionalButtonPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 20,
            20));
237       additionalButtonPanel.setBackground(new Color(236, 236, 236));
238       JButton additionalButton = additionalButtonPlugin.getAdditionalButton()
            ;
239       additionalButtonPanel.add(additionalButton);
240       container.add(additionalButtonPanel);
241     }
242
243
244
245   }
246
247   public JFrame getAMPFrame() {
248     return this.amp;
249   }
250
251   public MP3Player getMp3Player() {
252     return this.mp3Player;
253   }
254
255   public void setMp3Player(MP3Player player) {
256     this.mp3Player = player;
257   }
258
259   public JFileChooser getFc() {
260     return fc;
261   }
262
263   public void setFc(JFileChooser fc) {
264     this.fc = fc;
265   }
266
267   public JTextArea getInformation() {
268     return information;
269   }
```

```
270
271   public void setInformation(JTextArea information) {
272     this.information = information;
273   }
274
275   public Timer getPlayedTimer() {
276     return playedTimer;
277   }
278
279   public void setPlayedTimer(Timer playedTimer) {
280     this.playedTimer = playedTimer;
281   }
282 }
```

Listing A.2: Plugin-Implementierung des Features TimeProgress

```
1  package Plugins;
2
3  import Framework.AMPGui;
4  import Interfaces.ProgressIF;
5
6  public class TimeProgressPlugin implements ProgressIF{
7          AMPGui gui;
8
9          @Override
10         public String refreshStatus() {
11                 long totalSongLength = gui.getMp3Player().getTotalSongTime();
12                 long songPosition = gui.getMp3Player().getPlayedTime();
13                 String songTime = convertTimeToString(songPosition);
14                 String totalTime = convertTimeToString(totalSongLength);
15                 return songTime + " / " + totalTime;
16         }
17
18         public String convertTimeToString(long microsecs) {
19                 int seconds = (int) ((microsecs / (1000 * 1000)) % 60);
20                 int minutes = (int) (microsecs / (1000 * 1000 * 60));
21                 String secStr = (seconds < 10) ? "" + 0 + seconds : "" +
                       seconds;
22                 String minStr = (minutes < 10) ? "" + 0 + minutes : "" +
                       minutes;
23                 String timeAsStr = minStr + ":" + secStr;
24                 return timeAsStr;
25         }
26
27         @Override
28         public void registerGui(AMPGui gui) {
```

```
29                  this.gui = gui;
30          }
31
32  }
```

Listing A.3: Hilfsklasse zur Ausführung des Frameworks

```
1   package Framework;
2
3   import java.lang.reflect.InvocationTargetException;
4
5   import javax.swing.SwingUtilities;
6   import javax.swing.UIManager;
7   import javax.swing.UIManager.LookAndFeelInfo;
8
9   import Framework.AMPGui;
10  import Interfaces.AdditionalButtonIF;
11  import Interfaces.ProgressIF;
12  import Plugins.PercentProgressPlugin;
13  import Plugins.RestartSong;
14  import Plugins.TimeProgressPlugin;
15
16  public class FrameworkRunner {
17          public static void main(String[] args) {
18                  ProgressIF percentProgress = new PercentProgressPlugin();
19                  AdditionalButtonIF restartSong = new RestartSong();
20                  final AMPGui gui = new AMPGui(percentProgress);
21                  try {
22                          SwingUtilities.invokeAndWait(new Runnable() {
23                                  public void run() {
24                                          try {
25                                                  for (LookAndFeelInfo info :
                                                        UIManager.
                                                        getInstalledLookAndFeels()) {
26                                                      if ("Nimbus".equals(info.
                                                            getName())) {
27                                                          UIManager.setLookAndFeel(
                                                                info.getClassName());
28                                                          break;
29                                                      }
30                                                  }
31                                          } catch (Exception e) {
32                                                  // If Nimbus is not available,
                                                        you can set the GUI to
                                                        another look and feel.
33                                          }
```

```
34                                          gui.createAndShowGui();
                                                                }
35                          });
36                  } catch (InvocationTargetException e) {
37                          e.printStackTrace();
38                  } catch (InterruptedException e) {
39                          e.printStackTrace();
40                  }
41          }
42  }
```

Listing A.4: Hilfsklasse für die Attribute eines Musiktitels

```
1  package Framework;
2
3  import java.io.File;
4  import java.io.IOException;
5
6  import javax.sound.sampled.AudioSystem;
7  import javax.sound.sampled.Clip;
8  import javax.sound.sampled.LineUnavailableException;
9  import javax.sound.sampled.UnsupportedAudioFileException;
10
11 import com.mpatric.mp3agic.InvalidDataException;
12 import com.mpatric.mp3agic.Mp3File;
13 import com.mpatric.mp3agic.UnsupportedTagException;
14 /**
15  * This class is a container class for a clip which is used for the playlist
       in the MP3Player class
16  * @author Vasco
17  *
18  */
19 public class ClipInfo {
20         private Clip song;
21         private File songAsFile;
22         private Mp3File mp3;
23         private String interpret;
24         private String name;
25         private String album;
26         private String songlength;
27         private String songNumber;
28         private byte[] imageData;
29
30         public ClipInfo(String songToLoad) throws
               UnsupportedAudioFileException, IOException,
               UnsupportedTagException, InvalidDataException,
```

```
            LineUnavailableException {
31              File file = new File(songToLoad);
32              songAsFile = file;
33              mp3 = new Mp3File(songToLoad);
34
35              //Setting the mp3 title and interpret
36              if(mp3.hasId3v2Tag()) {
37                      interpret = mp3.getId3v2Tag().getArtist();
38                      name = mp3.getId3v2Tag().getTitle();
39                      album = mp3.getId3v2Tag().getAlbum();
40                      songNumber = mp3.getId3v2Tag().getTrack();
41                      if(mp3.getId3v2Tag().getAlbumImage() != null) {
42                              setImageData(mp3.getId3v2Tag().getAlbumImage
                                    ());
43                      }
44              }
45              else if(mp3.hasId3v1Tag()) {
46                      interpret = mp3.getId3v1Tag().getArtist();
47                      name = mp3.getId3v1Tag().getTitle();
48                      album = mp3.getId3v1Tag().getAlbum();
49                      songNumber = mp3.getId3v1Tag().getTrack();
50
51              }
52              else {
53                      interpret = "Unknown artist";
54                      name = "Unknown title";
55                      album = "Unknown album";
56                      songNumber = "0";
57              }
58
59              //Exercise3
60
61
62              song = AudioSystem.getClip();
63              songlength = convertToTime(mp3.getLengthInSeconds());
64
65      }
66
67      public Clip getSong() {
68              return song;
69      }
70      public void setSong(Clip song) {
71              this.song = song;
72      }
73      public Mp3File getMp3() {
74              return mp3;
```

```
75          }
76          public void setMp3(Mp3File mp3) {
77                  this.mp3 = mp3;
78          }
79          public String getInterpret() {
80                  return interpret;
81          }
82          public void setInterpret(String interpret) {
83                  this.interpret = interpret;
84          }
85          public String getName() {
86                  return name;
87          }
88          public void setName(String name) {
89                  this.name = name;
90          }
91          public String getAlbum() {
92                  return album;
93          }
94          public void setAlbum(String album) {
95                  this.album = album;
96          }
97          public String getSonglength() {
98                  return songlength;
99          }
100         public void setSonglength(String songlength) {
101                 this.songlength = songlength;
102         }
103
104         public byte[] getImageData() {
105                 return imageData;
106         }
107
108         public void setImageData(byte[] imageData) {
109                 this.imageData = imageData;
110         }
111
112         public String infoForListWidget() {
113                 return songNumber + "␣␣␣:␣␣␣" + name + "␣|␣" + interpret + "␣
                        |␣"  + album + "␣|␣" + songlength;
114         }
115
116         /**
117          * Helper method which converts microsecs to minutes and seconds
118          * @param microsecs
119          * @return String
```

```
120          */
121         public String convertToTime(long seconds) {
122                 int minutes = (int) (seconds / 60);
123                 int secs = (int) (seconds % 60);
124                 String secStr = (secs < 10) ? "" + 0 + secs : "" + secs;
125                 String minStr = (minutes < 10) ? "" + 0 + minutes : "" +
                        minutes;
126                 String tmp = minStr + ":" + secStr;
127                 return tmp;
128         }
129
130         public File getSongAsFile() {
131                 return songAsFile;
132         }
133
134         public void setSongAsFile(File songAsFile) {
135                 this.songAsFile = songAsFile;
136         }
137
138         public String getSongNumber() {
139                 return songNumber;
140         }
141
142         public void setSongNumber(String songNumber) {
143                 this.songNumber = songNumber;
144         }
145 }
```

Listing A.5: Klasse des Mp3-Player-Objekts

```
1  package Framework;
2
3  import java.io.IOException;
4
5  import javax.sound.sampled.AudioFormat;
6  import javax.sound.sampled.AudioInputStream;
7  import javax.sound.sampled.AudioSystem;
8  import javax.sound.sampled.Clip;
9  import javax.sound.sampled.LineUnavailableException;
10 import javax.sound.sampled.UnsupportedAudioFileException;
11
12 import com.mpatric.mp3agic.InvalidDataException;
13 import com.mpatric.mp3agic.Mp3File;
14 import com.mpatric.mp3agic.UnsupportedTagException;
15
16 import javazoom.jl.decoder.JavaLayerException;
```

```
17 import Framework.ClipInfo;
18
19 /**
20  * A simple MP3Player, which is able to play an .mp3-file and pause, resume,
     and
21  * stop the playback.
22  */
23 public class MP3Player {
24         public final static int STATUS_READY = 0;
25         public final static int STATUS_PLAYING = 1;
26         public final static int STATUS_PAUSED = 2;
27         public final static int STATUS_FINISHED = 3;
28
29         private int playerStatus = STATUS_READY;
30         private AudioInputStream audioStream;
31         private Clip clip;
32         private String title;
33         private String interpret;
34         private byte[] imageData;
35
36         public MP3Player() {
37
38         }
39
40         /**
41          * Creates a new MP3Player from given InputStream.
42          *
43          * @param filename
44          * @throws IOException
45          * @throws UnsupportedAudioFileException
46          * @throws LineUnavailableException
47          * @throws InvalidDataException
48          * @throws UnsupportedTagException
49          */
50         public MP3Player(String filename) throws
             UnsupportedAudioFileException,
51                         IOException, LineUnavailableException,
                             UnsupportedTagException,
52                         InvalidDataException {
53                 ClipInfo newSong = new ClipInfo(filename);
54                 setClip(newSong);
55                 Mp3File mp3 = new Mp3File(filename);
56
57                 // Setting the mp3 title and interpret
58                 if (mp3.hasId3v2Tag()) {
59                         interpret = mp3.getId3v2Tag().getArtist();
```

```
60                          title = mp3.getId3v2Tag().getTitle();
61                          if (mp3.getId3v2Tag().getAlbumImage() != null) {
62                                  setImageData(mp3.getId3v2Tag().getAlbumImage
                                        ());
63                          }
64                  } else if (mp3.hasId3v1Tag()) {
65                          interpret = mp3.getId3v1Tag().getArtist();
66                          title = mp3.getId3v1Tag().getTitle();
67
68                  } else {
69                          interpret = "Unknown artist";
70                          title = "Unknown title";
71                  }
72          }
73
74          /**
75           * This methods resets the clip object to a new one of the file with
                the
76           * given file name NOTE: Not used at the moment
77           *
78           * @param filename
79           * @throws UnsupportedTagException
80           * @throws InvalidDataException
81           * @throws IOException
82           * @throws UnsupportedAudioFileException
83           * @throws LineUnavailableException
84           */
85          public void setClip(ClipInfo newClip) throws UnsupportedTagException,
86                          InvalidDataException, IOException,
                                UnsupportedAudioFileException,
87                          LineUnavailableException {
88                  if (clip != null) {
89                          reset();
90                  }
91                  if(newClip != null) {
92                  clip = newClip.getSong();
93                  setImageData(newClip.getImageData());
94                  interpret = newClip.getInterpret();
95                  title = newClip.getName();
96                  audioStream = AudioSystem.getAudioInputStream(newClip.
                        getSongAsFile());
97                  AudioFormat baseFormat = audioStream.getFormat();
98                  AudioFormat decodedFormat = new AudioFormat(
99                                  AudioFormat.Encoding.PCM_SIGNED, baseFormat.
                                        getSampleRate(),
```

```
100                                 16, baseFormat.getChannels(), baseFormat.
                                        getChannels() * 2,
101                                 baseFormat.getSampleRate(), false);
102                 AudioInputStream stream = AudioSystem.getAudioInputStream(
103                                 decodedFormat, audioStream);
104                 clip.open(stream);
105                 }
106         }
107
108         /**
109          * Starts playback (resumes if paused) (restarts if finished).
110          *
111          * @throws JavaLayerException
112          *             when there is a problem decoding the file.
113          */
114         public void play() {
115                 synchronized (clip) {
116                         if (clip != null && clip.isOpen() && clip.
                                getMicrosecondLength() == clip.
                                getMicrosecondPosition()) {
117                                 clip.stop();
118                                 clip.setFramePosition(0);
119                                 clip.start();
120                                 playerStatus = STATUS_PLAYING;
121                         }
122                         else if (clip != null && clip.isOpen()) {
123                                 clip.start();
124                                 playerStatus = STATUS_PLAYING;
125                         }
126                 }
127         }
128
129         /**
130          * Pauses playback.
131          */
132         public void pause() {
133                 synchronized (clip) {
134                         if (playerStatus == STATUS_PLAYING) {
135                                 clip.stop();
136                                 playerStatus = STATUS_PAUSED;
137                         }
138                 }
139         }
140
141         /**
142          * Resumes playback.
```

```
143         */
144         public void resume() {
145                 synchronized (clip) {
146                         if (playerStatus == STATUS_PAUSED) {
147                                 clip.start();
148                                 playerStatus = STATUS_PLAYING;
149                         }
150                 }
151         }
152 
153         /**
154          * Stops playback. If not playing, does nothing.
155          */
156         public void stop() {
157                 synchronized (clip) {
158                         clip.stop();
159                         clip.setFramePosition(0);
160                         playerStatus = STATUS_FINISHED;
161                 }
162         }
163 
164         public void restart() {
165                 synchronized (clip) {
166                         clip.stop();
167                         clip.setFramePosition(0);
168                         clip.start();
169                         playerStatus = STATUS_PLAYING;
170                 }
171         }
172 
173         /**
174          * Returns the current player status.
175          *
176          * @return the player status
177          */
178         public int getPlayerStatus() {
179                 return this.playerStatus;
180         }
181 
182         /**
183          * Closes the player, regardless of current state.
184          */
185         public void close() {
186                 synchronized (clip) {
187                         if (clip != null && clip.isOpen()) {
188                                 clip.stop();
```

```
189                                 clip.close();
190                         }
191                 }
192                 try {
193                         audioStream.close();
194                         audioStream = null;
195                 } catch (IOException e) {
196                         // we are terminating, thus ignore exception
197                 }
198         }
199
200         public String getID3InformationWithoutProgress() {
201                 if (clip.isOpen()) {
202                         return title + "␣|␣" + ((interpret==null) ? "No␣
                                Interpret" : interpret);
203                 } else {
204                         return "No␣file␣loaded";
205                 }
206         }
207
208         /**
209          * Checks if the clip is open
210          *
211          * @return boolean
212          */
213         public boolean isReady() {
214                 if (clip.isOpen()) {
215                         return true;
216                 } else {
217                         return false;
218                 }
219         }
220
221         public long getPlayedTime() {
222                 return clip.getMicrosecondPosition();
223         }
224
225         public long getTotalSongTime() {
226                 return clip.getMicrosecondLength();
227         }
228
229         /**
230          * Resets all current resources so that a new one could be allocated
231          */
232         public void reset() {
233                 clip.stop();
```

```
234                 clip.close();
235                 clip.setFramePosition(0);
236                 this.clip = null;
237                 this.audioStream = null;
238         }
239
240         public byte[] getImageData() {
241                 return imageData;
242         }
243
244         public void setImageData(byte[] imageData) {
245                 this.imageData = imageData;
246         }
247 }
```

Literaturverzeichnis

[1] Website der „Carnegie Mellon Website“ zu Software Product Lines. Stand 14.08.2015.

[2] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. Family-Based Model Checking without a Family-Based Model Checker. *Springer International Publishing*, 2015.

[3] Aleksandar S. Dimovski, Claus Brabrand und Andrzej Wąsowski. Variability Abstractions: Trading Precision for Speed in Family-Based Analyses (Extended Version). *CoRR*, abs/1503.04608, 2015.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. *Compiler: Prinzipen, Techniken und Werkzeuge*. Pearson Studium, 2008.

[5] Alvin Ahnassay, Ebrahim Bagheri and Dragan Gasevic. Empirical evaluation in software product line engineering, 2012.

[6] Amir Molzam Sharifloo. *Requirements Verification of Variability-Intensive Systems*. PhD thesis, Politecnico di Milano, 2014.

[7] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay and Pierre-Yves Schobbens. Formal Semantics, Modular Specification, and Symbolic Verification of Product-Line Behaviour. *Science of Computer Programming, Volume 80, Part B, Pages 416 - 439*, 2014.

[8] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay and Jean-François Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *Software Engineering, IEEE Transactions on*, 39(8):1069–1089, Aug 2013.

[9] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay. Symbolic model checking of software product lines. *ICSE '11 Proceedings of the 33rd International Conference on Software Engineering, Pages 321-330*, 2011.

[10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay and Jean François Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 335–344, New York, NY, USA, 2010. ACM.

[11] Andy Kenner, Christian Kästner, Steffen Haase and Thomas Leich. TypeChef: Toward Type Checking #ifdef Variability in C. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 25–32, New York, NY, USA, 2010. ACM.

[12] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] Benjamin Delaware, William Cook and Don Batory. Product Lines of Theorems. *SIGPLAN Not.*, 46(10):595–608, Oct. 2011.

[14] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007.

[15] Carlo Ghezzi and Amir Molzam Sharifloo. Model-based Verification of Quantitative Non-functional Properties for Software Product Lines. *Inf. Softw. Technol.*, 55(3):508–524, Mar. 2013.

[16] Chang Hwan Peter Kim, Don S. Batory and Sarfraz Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 57–68, New York, NY, USA, 2011. ACM.

[17] Chang Hwan Peter Kim, Eric Bodden, Don Batory and Sarfraz Khurshid. Reducing Configurations to Monitor in a Software Product Line. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 285–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[19] Christian Kästner, Klaus Ostermann and Sebastian Erdweg. A Variability-aware Module System. *SIGPLAN Not.*, 47(10):773–792, Oct. 2012.

[20] Christian Kästner, Sven Apel, Thomas Thüm and Gunter Saake. Type Checking Annotation-based Product Lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14:1–14:39, July 2012.

[21] Christioh Kern and Mark R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, Apr. 1999.

[22] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell and Anders Wesslén. *Experimentation in Software Engineering*. Springer-Verlag Berlin Heidelberg, 2012.

[23] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo and Paulo Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 13–24, New York, NY, USA, 2012. ACM.

[24] Claus Hunsen. Experiments on Type Checking of Software Product Lines. Master's thesis, Universität Passau, 2013.

[25] Clemens Dubslaff, Christel Baier and Sascha Klüppelholz. Probabilistic Model Checking for Feature-Oriented Systems. *Transactions on Aspect-Oriented Software Development XII*, 2015.

[26] Clemens Dubslaff, Sascha Klüppelholz and Christel Baier. Probabilistic Model Checking for Energy Analysis in Software Product Lines. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 169–180, New York, NY, USA, 2014. ACM.

[27] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba and Mira Mezini. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. *SIGPLAN Not.*, 48(6):355–364, June 2013.

[28] Genaína N. Rodrigues, Vander Alves, Vinicius Nunes, André Lanna, Maxime Cordy, Pierre-Yves Schobbens, Amir Molzam Sharifloo and Axel Legay. Modeling and Verification for Probabilistic Properties in Software Product Lines. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*, pages 173–180, Jan 2015.

[29] Hamideh Sabouri and Ramtin Khosravi. Efficient Verification of Evolving Software Product Lines. In *Proceedings of the 4th IPM International Conference on Fundamentals of Software Engineering*, FSEN'11, pages 351–358, Berlin, Heidelberg, 2012. Springer-Verlag.

[30] Hamideh Sabouri and Ramtin Khosravi. Modeling and Verification of Reconfigurable Actor Families. *Journal of Universal Computer Science, Vol. 19, No. 2 (2013), Pages 207-232*, 2013.

[31] Hamideh Sabouri and Ramtin Khosravi. Reducing the Verification Cost of Evolving Product Families Using Static Analysis Techniques. *Sci. Comput. Program.*, 83:35–55, Apr. 2014.

[32] Hendrik Post and Carsten Sinz. Configuration Lifting: Verification Meets Software Configuration. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 347–350, Washington, DC, USA, 2008. IEEE Computer Society.

[33] Ina Schaefer, Dilian Gurov and Siavash Soleimanifard. Compositional Algorithmic Verification of Software Product Lines. *Int'l Symposium on Formal Methods for Components and Objects (FMCO)*, 2010.

[34] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand und Andrzej Wąsowski. Systematic Derivation of Correct Variability-aware Program Analyses. *Sci. Comput. Program.*, 105(C):145–170, July 2015.

[35] Jean-Marc Jézéquel and Bertrand Meyer. Design by Contract: The Lessons of Ariane. *Computer*, 30(1):129–130, Jan 1997.

[36] Jens Meinicke. JML-Based Verification for Feature-Oriented Programming. BachelorsThesis, 2013.

[37] Jiangfan Shi, Myra B. Cohen and Matthew B. Dwyer. Integration Testing of SPLs Using Compositional Symbolic Excecution. *Lecture Notes in Computer Science*, 2012.

[38] Jing Liu, Samik Basu and Robyn R. Lutz. Compositional Model Checking of Software Product Lines Using Variation Point Obligations. *Automated Software Engg.*, 18(1):39–76, Mar. 2011.

[39] Johann M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer-Verlag Berlin Heidelberg, 2001.

[40] John Greenyer, Amir Molzam Sharifloo, Maxime Cordy and Patrick Heymans. Features Meet Scenarios: Modeling and Consistency-checking Scenario-based Product Line Specifications. *Requir. Eng.*, 18(2):175–198, June 2013.

[41] Jörg Liebig. *Analysis and Transformation of Configurable Systems*. PhD thesis, Universität Passau, 2015.

[42] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre und Christian Lengauer. Large-Scale Variability Aware Type Checking and Dataflow Analysis. Technical report, 2012.

[43] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre und Christian Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 81–91, New York, NY, USA, 2013. ACM.

[44] Jürgen Hedderich and Lothar Sachs. *Angewandte Statistitk - Methodensammlung mit R.* Springer-Verlag Berlin Heidelberg, 2012.

[45] Kim Lauenroth, Klaus Pohl and Simon Töhning. Model Checking of Domain Artifacts in Product Line Engineering. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 269–280, Nov 2009.

[46] Klaus Pohl, Günter Böckle and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* 2005.

[47] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 211–220, New York, NY, USA, 2006. ACM.

[48] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. tech. rep. cmu/sei-90-tr-21. Technical report, Software Engineering Institute - Carnegie Mellon University, 1990.

[49] Leopoldo Teixeira, Paulo Borba and Rohit Gheyi. Safe Composition of Configuration Knowledge-Based Software Product Lines. In *Software Engineering (SBES), 2011 25th Brazilian Symposium on*, pages 263–272, Sept 2011.

[50] Malte Lochau, Stephan Mennicke, Hauke Baller and Lars Ribbeck. Incremental Model Checking of Delta-Oriented Software Product Lines. *Journal of Logical and Algebraic Methods in Programming*, 85(1, Part 2):245 – 267, 2016. Formal Methods for Software Product Line Engineering.

[51] Martin Eisend. Metaanalyse - Einführung und kritische Diskussion. 2004.

[52] Martin Kuhlemann, Don Batory and Christian Kästner. Safe Composition of Non-monotonic Features. *SIGPLAN Not.*, 45(2):177–186, Oct. 2009.

[53] Matthias Praast. Effiziente Kodierung von Variabilität in Spezifikationen. Master's thesis, Otto-von-Guericke-Universität Magdeburg, 2014.

[54] Mauricio Alférez, Roberto E. Lopez-Herrejon, Ana Moreira, Vasco Amaral and Alexander Egyed. Supporting Consistency Checking Between Features and Software Product Line Use Scenarios. In *Proceedings of the 12th International Conference on Top Productivity Through Software Reuse*, ICSR'11, pages 20–35, Berlin, Heidelberg, 2011. Springer-Verlag.

[55] Maxime Cordy, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans and Axel Legay. Simulation-based Abstractions for Software Product-line Model Checking. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 672–682, Piscataway, NJ, USA, 2012. IEEE Press.

[56] Maxime Cordy, Patrick Heymans, Axel Legay, Pierre-Yves Schobbens, Bruno Dawagne and Martin Leucker. Counterexample Guided Abstraction Refinement of Product-line Behavioural Models. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 190–201, New York, NY, USA, 2014. ACM.

[57] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans and Axel Legay. Behavioural Modelling and Verification of Real-time Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 66–75, New York, NY, USA, 2012. ACM.

[58] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans and Axel Legay. Beyond Boolean Product-line Model Checking: Dealing with Feature Attributes and Multi-features. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 472–481, Piscataway, NJ, USA, 2013. IEEE Press.

[59] Paola Rodrigues Godoy Accioly. Comparing Different Testing Strategies for Software Product Lines. Master's thesis, Universidade Federal de Pernambuco, 2012.

[60] Patrick Cousot and Radhia Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In J. Esparza, O. Grumberg, and M. Broy, editors, *Logics and Languages for Reliability and Security*, NATO Science Series III: Computer and Systems Sciences, pages 1–29. IOS Press, 2010.

[61] Peter Lutz. Typprüfung von Produktlinien in Fuji. Master's thesis, Universität Passau, 2013.

[62] Sahil Thaker, Don Batory, David Kitchin and William Cook. Safe Composition of Product Lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 95–104, New York, NY, USA, 2007. ACM.

[63] Samireh Jalali and Claes Wohlin. Systematic literature studies: Database searches vs. backward snowballing. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 29–38, New York, NY, USA, 2012. ACM.

[64] Sara Bessling und Michaela Huhn. Towards Formal Safety Analysis in Feature-Oriented Product Line Development. In *Foundations of Health Information Engineering and Systems*, pages 217–235. Springer, 2014.

[65] Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen and Sven Apel. A Comparison of Product-based, Feature-based, and Family-based Type Checking. *SIGPLAN Not.*, 49(3):115–124, Oct. 2013.

[66] Sheng Chen, Martin Erwig und Eric Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. Program. Lang. Syst.*, 36(1):1:1–1:54, Mar. 2014.

[67] Sheng Chen und Martin Erwig. Type-based Parametric Analysis of Program Families. *SIGPLAN Not.*, 49(9):39–51, Aug. 2014.

[68] Shoham Ben-David, Baruch Sterin, Joanne M. Atlee und Sandy Beidu. Symbolic Model Checking of Product-line Requirements Using SAT-based Methods. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 189–199, Piscataway, NJ, USA, 2015. IEEE Press.

[69] Sönke Holthusen. Typsicherheit in Feature-orientierten Software-Produktlinien in Feature-IDE. Master's thesis, Otto-von-Guericke-Universität Magdeburg, 2012.

[70] Stephen Jeff Rustenbach. *Meta-Analyse: Eine anwendungsorientierte Einführung.* 2003.

[71] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger and Dirk Beyer. Strategies for Product-line Verification: Case Studies and Experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 482–491, Piscataway, NJ, USA, 2013. IEEE Press.

[72] Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology Vol. 8*, 2009.

[73] Sven Apel, Don Batory, Christian Kästner and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer Verlag, 2013.

[74] Sven Apel, Hendrik Speidel, Philip Wendler, Alexander von Rhein and Dirk Beyer. Detection of Feature Interactions Using Feature-aware Verification. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 372–375, Washington, DC, USA, 2011. IEEE Computer Society.

[75] Tessa Belder, Maurice H. ter Beek and Erik P. de Vink. Coherent branching feature bisimulation. *In Proceedings FMSPLE 2015*, 2015.

[76] Thomas Thüm, Ina Schaefer, Martin Kuhlemann and Sven Apel. Proof Composition for Deductive Verification of Software Product Lines. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 270–277, March 2011.

[77] Thomas Thüm, Ina Schaefer, Sven Apel und Martin Hentschel. Family-based Deductive Verification of Software Product Lines. *SIGPLAN Not.*, 48(3):11–20, Sept. 2012.

[78] Thomas Thüm, Jens Meinicke, Fabian Benduhn, Martin Hentschel, Alexander von Rhein and Gunter Saake. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 177–186, New York, NY, USA, 2014. ACM.

[79] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.