

Otto-von-Guericke-Universität Magdeburg



OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG



FAKULTÄT FÜR
INFORMATIK

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Statische Referenzanalyse in C-Präprozessor-konfigurierten Anwendungen

Verfasser:

Andy Kenner

30.08.2010

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,

Dr.-Ing. Christian Kästner,

Dipl.-Wirtsch.-Inf. Thomas Leich

Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Postfach 4120, D-39016 Magdeburg

Kenner, Andy

*Statische Referenzanalyse in C-Präprozessor-
konfigurierten Anwendungen*

Diplomarbeit, Otto-von-Guericke-Universität
Magdeburg, 2010.

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich auf dem Weg durch das Studium begleitet haben. Ein besonderer Dank gilt dabei:

- Prof. Dr. rer. nat. habil. Gunter Saake, Dipl.-Wirtsch.-Inf. Thomas Leich und speziell Dr.-Ing. Christian Kästner für die intensive Betreuung meiner Diplomarbeit und die zahlreichen Diskussionen sowie Ratschläge und Anmerkungen.
- meinen Kommilitonen Enrico Herrmann, Martin Sturm, Martin Paatz und Bernd Zuther für die gegenseitige Unterstützung in allen Etappen des Studiums.
- meiner Familie, die dieses Studium erst ermöglicht haben und mir die gesamte Zeit über den Rücken gestärkt haben.
- Daniela Huhn und Katrin Barkholz für kritische Anmerkungen zu Form und Inhalt sowie Rechtschreibung und Grammatik.

Ein abschließender Dank gilt Dr.-Ing. Christian Kästner, Dipl.-Wirtsch.-Inf. Thomas Leich und Steffen Haase, die es ermöglicht haben, die Ergebnisse dieser Diplomarbeit in einem Paper für den FOSD Workshop zu präsentieren.

Aufgabenstellung

Eine Softwareproduktlinie (SPL) beschreibt eine Menge von Anwendungen, die mit einer gemeinsamen Quellcodebasis erschaffen werden. Die Anwendungen besitzen Gemeinsamkeiten und Unterschiede, die durch Merkmale beschrieben werden. Je nach der Auswahl der in einem bestimmten Produkt enthaltenen Funktionalitäten werden die dazugehörigen Merkmale kombiniert und eine Variante der SPL generiert. Die Variantenanzahl kann exponentiell mit der Anzahl der Merkmale ansteigen. Daher ist es nicht realistisch alle Varianten isoliert auf Korrektheit zu untersuchen, sondern es werden neue Mechanismen gebraucht, welche die gesamte SPL auf einmal prüfen.

Eine Klasse von Fehlern, genannt Typfehler, kann statisch von einem Typsystem der Sprache erkannt werden. Dazu gehören etwa Aufrufe von nicht definierten Funktionen. Die aktuelle Forschung hat Prinzipien produktlinienorientierter Typsysteme, welche die gesamte Produktlinie in einem Schritt prüfen, aufgezeigt und für verschiedene Implementierungsmechanismen zur Verfügung gestellt.

In der Praxis wird ein Grossteil der Variabilität in Softwareproduktlinien mit Annotationstechniken wie dem C-Präprozessor erzeugt. Ziel dieser Arbeit ist es ein produktlinienorientiertes Typsystem für C zu entwickeln bzw. zu untersuchen inwieweit eine solche Entwicklung möglich ist. Der Fokus liegt auf dem Prüfen von Referenzen im Quelltext, wie Variablenzugriffe und Methodenaufrufe. Dabei soll insbesondere untersucht werden, inwieweit Rückwärtskompatibilität zum C-Präprozessor möglich ist, so dass auch existierende SPL-Implementierungen geprüft werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis	ix
Tabellenverzeichnis	xi
Abkürzungsverzeichnis	xiii
1 Einleitung	1
1.1 Motivation	1
1.2 Ziel der Arbeit	3
1.3 Beitrag der Arbeit	4
1.4 Gliederung der Arbeit	5
2 Grundlagen	7
2.1 C-Präprozessor	7
2.1.1 Einfügen von Dateiinhalten	7
2.1.2 Definition von Makros und symbolischen Konstanten	8
2.1.3 Bedingte Übersetzung	9
2.1.4 Weitere Direktiven	10
2.2 Softwareproduktlinien	11
2.2.1 Domänen- und Anwendungsentwicklung	12
2.2.2 Softwareproduktlinienimplementierung	13
2.3 Merkmalmodellierung	16
2.3.1 Transformation von Merkmaldiagrammen	17
2.3.2 Erweiterung von Merkmaldiagrammen	19
2.4 Typsysteme	20
3 Konzept für die CPP-spezifische Analyse	25
3.1 Partieller C-Präprozessor	27
3.1.1 Beispiel	28
3.1.2 Include Guards	28
3.2 Disziplintest	29
3.2.1 Disziplinierte Annotationen	30
3.2.2 Undisziplinierte Annotationen	33
3.2.3 Manuelle Disziplinierung von Annotationen	33
3.3 Erweiterter Parser	34
3.4 Referenzanalyse	35
3.4.1 Referenzen ermitteln	35
3.4.2 Referenzpaare ermitteln	36
3.4.3 Notwendigkeit der Analyse	37

3.4.4	Beispiele	38
3.5	CSP-Transformation und -Auswertung	40
3.5.1	Gestalt der Hierarchieebenen	40
3.5.2	Gesamtbedingung für die Präsenz eines Quellcodeelementes	41
3.5.3	Auftreten eines Typfehlers	43
3.5.4	Überführung in ein Lösungssystem	44
3.6	Kombiniertes Analysekonzept	45
3.7	Kapitelzusammenfassung	46
4	Umsetzung der statischen Referenzanalyse	49
4.1	Partieller C-Präprozessor	49
4.2	Präprozessorerweiterung des C-Parsers	50
4.2.1	ANother Tool for Language Recognition - ANTLR	52
4.2.2	C-Grammatik und Präprozessorvariabilität	52
4.2.3	Erweiterter Parser	56
4.3	Informationsaufbereitung für Referenzpaare	56
4.3.1	Traversierung des AST	57
4.3.2	Disziplinierte Annotationen	57
4.3.3	Ermittlung der Direktivenhierarchie	58
4.3.4	Befüllung der Symboltabelle	59
4.3.5	Bildung von Referenzpaaren	63
4.4	CSP-Transformation der Annotationen	64
4.4.1	Überführung eines Direktivenausdruckes	65
4.4.2	Konstruktion der Hierarchiebedingungen	67
4.5	Formulierung des Bedingungserfüllungsproblems	68
4.6	Vollständige Analyse eines Projektes	69
4.7	Kapitelzusammenfassung	70
5	Evaluation	71
5.1	Statische Referenzanalyse in boa	71
5.2	Partieller C-Präprozessor	72
5.3	Disziplin der Annotationen	72
5.4	Ermittlung von Typfehlern	74
5.4.1	Funktion	74
5.4.2	Variable	75
5.4.3	Struktur-/ Union-Typ	75
5.4.4	Aufzählungstyp	76
5.4.5	Typdefinition	76
5.4.6	Gesamtergebnis	76
5.5	Detaillierte Fehlerbetrachtung	77
5.5.1	Typfehler durch fehlende Zuordnung	77
5.5.2	Typfehler durch Fehlkonfiguration	79
5.5.3	Bestätigte Fehler	81
5.6	Unterstützte und analysierbare CPP-Konditionen	82
5.7	Kapitelzusammenfassung	84

6 Probleme und Perspektiven	85
6.1 Nicht nachvollziehbare Elemente	85
6.2 Partieller C-Präprozessor	87
6.2.1 Plattformspezifische Konfiguration	87
6.2.2 Konfiguration von Direktiven	87
6.2.3 Perspektiven des partiellen Präprozessors	88
6.3 Alternativer C-Quellcode	89
6.4 Perspektiven für ein produktlinienorientiertes Typsystem	90
6.5 Kapitelzusammenfassung	90
7 Verwandte Arbeiten	91
7.1 Analyse von C und CPP	91
7.2 Produktlinienorientierte Typsysteme	93
8 Zusammenfassung	97
Literaturverzeichnis	101

Abbildungsverzeichnis

2.1	Syntax und Anwendungsbeispiel für die Dateinklusion	8
2.2	Syntax und Anwendungsbeispiel einer Makrodefinition	8
2.3	Syntax und Anwendungsbeispiel einer Makrodefinition als symbolische Konstante	9
2.4	Syntax und Anwendungsbeispiel für die bedingte Übersetzung	10
2.5	Spezialformen der bedingten Übersetzung	10
2.6	Domänen- und Anwendungsentwicklung	13
2.7	Notation der Beziehungen in Merkmaldiagrammen	17
2.8	Erweitertes Merkmaldiagramm	20
3.1	Übersicht der TypeChef-Phasen	25
3.2	Ursprungszustand des Quellcodes	28
3.3	Ergebnis des partiellen Präprozessors	28
3.4	Einfügeschutz eines Header-Files (Femto OS)	29
3.5	Annotation von globalen Elementen (Femto OS)	31
3.6	Annotation von lokalen Elementen (Femto OS)	32
3.7	Undisziplinierte Annotationen	33
3.8	Erweiterung des AST um CPP-Direktiven	35
3.9	Fehlerhafte Referenzen durch Variabilität 1 (Femto OS)	39
3.10	Fehlerhafte Referenzen durch Variabilität 2 (Femto OS)	39
3.11	Hierarchie in Präprozessordirektiven (Femto OS)	43
4.1	Herauslösen der Konfigurationsdirektiven	50
4.2	Hauptregel für die Verzweigung in die CPP-Direktiven	53
4.3	Grammatikdefinition für die Direktive <code>#ifdef</code>	54
4.4	Grammatikdefinition für die Direktive <code>#ifndef</code>	54
4.5	Grammatikdefinition für die Direktiven <code>#if</code> , <code>#elif</code> , <code>#else</code>	55
4.6	Grammatikdefinition für die Direktive <code>#endif</code>	55
4.7	Quellcodeauszug zum AST Beispiel 1 (Femto OS)	60
4.8	AST Beispiel 1 (Femto OS)	60
4.9	Quellcodeauszug und AST Beispiel 2 (Femto OS)	62
5.1	Beispiel für eine die Disziplin verletzende Annotation (boa)	73
5.2	Ergebnis der manuellen Transformation	73
5.3	Manuelle Fehleranalyse - fehlende Zuordnung 1	78
5.4	Manuelle Fehleranalyse - fehlende Zuordnung 2	78
5.5	Manuelle Fehleranalyse - fehlende Zuordnung 3	78
5.6	Manuelle Fehleranalyse - Konditionenvergleich 1	79
5.7	Manuelle Fehleranalyse - Konditionenvergleich 2	80

5.8	Relationale Konditionen (gretl)	82
5.9	Weitere durch TypeChef untersuchbare Konditionen	83

Tabellenverzeichnis

2.1	Weitere Direktiven des CPP	11
2.2	Transformationsregeln für Merkmaldiagramme in aussagenlogische Ausdrücke und CSP Ausdrücke	18
2.3	Klassifizierung von Programmiersprachen	22
3.1	Erweiterte Symboltabelle	36
4.1	Neue Einträge der erweiterten Symboltabelle 1	61
4.2	Neue Einträge der erweiterten Symboltabelle 2	62
4.3	Beziehungen zwischen den Erstellungs- und Zugriffsoperationen eines Elementes	63
5.1	Zusammenfassung der Referenzanalyse für das Projekt boa	77

Abkürzungsverzeichnis

ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
CPP	C-Präprozessor
CSP	Constraint Satisfaction Problem
SAT	Boolean SATisfiability Problem
SoC	Separation of Concerns
SPL	Softwareproduktlinie

KAPITEL 1

Einleitung

1.1 Motivation

Die aktuelle Softwareentwicklung muss immer höheren Ansprüchen genügen. Der steigende Bedarf an die zu realisierenden Funktionalitäten bewirkt einen Anstieg der Komplexität der gesamten Software. Folglich muss eine Anwendung mit einer hohen Produktivität und Qualität entwickelt werden, die zudem einfach zu warten und effektiv sowie leicht zu erweitern ist.

Das Prinzip der Wiederverwendung stellt eine Möglichkeit dar, die oben aufgeführten Anforderungen einzuhalten. Die Produktlinientechnologie wendet dieses Prinzip erfolgreich in der Industrie an [PBL05]. Hierbei werden Fabrikate aus wiederverwendeten Komponenten zusammengestellt.

Die Entwicklung einer Software auf der Basis einer Softwareproduktlinie (SPL) ist von diesem Vorgehen inspiriert und hat das Ziel, die einzelnen Artefakte einer Anwendung zu kapseln und in neuen Kombinationen wiederzuverwenden [PBL05]. Ein aus einer SPL entstehendes Produkt ist auf die Anforderung einer Gruppe von Nutzern ausgerichtet und maßgeschneidert. Nach POHL et al. [PBL05] wird die Idee verfolgt, die bisherigen Methoden Massensoftware und Individualsoftware zu vereinen und von den Vorteilen dieser beiden Strategien zu profitieren. Die Umsetzung eines speziellen Systems wird nicht individuell vollzogen, sondern für einen Anwenderbereich (Domäne) anhand der darin benötigten funktionellen Eigenschaften geplant und in einer gemeinsamen Quellcodebasis realisiert. Die Funktionalität wird mit Hilfe von Merkmalen beschrieben, um die verschiedenen Programmcharakteristiken zu abstrahieren. Die Auswahl einer bestimmten Menge von Merkmalen legt eine konkrete Variante der SPL fest. Im Gegensatz zu einer Massensoftware, die eine allgemeine Funktionalität für einen großen Anwenderbereich bereitstellt, werden die besonderen Bedürfnisse einer Nutzergruppe beachtet und die Auswahl hinsichtlich der jeweiligen Anforderungen zugeschnitten, ohne eine Anwendung in Form einer Individualsoftware wiederholt neu umzusetzen [CN01]. Eine SPL wird mittels einer gemeinsamen Quellcodebasis entwickelt und setzt sich aus Artefakten zusammen, die bei jeder Generierung einer Variante wiederverwendet werden. Daraus resultiert eine fortlaufende Verbesserung und Qualitätssteigerung der gesamten SPL, wobei gleichzeitig die Entwicklungszeit und -kosten gesenkt werden [Wit96, CN01].

Für die Implementierung der Artefakte beziehungsweise der SPL stehen verschiedene Techniken zur Verfügung. Diese werden nach KÄSTNER et al. [KAK08] in Kompositions- und Annotationsansätze unterteilt. Die Kompositionsansätze nutzen physisch voneinander getrennte Einheiten zur Umsetzung der merkmalspezifischen Artefakte. Durch

den Vorgang der Komposition werden die vorgefertigten Einheiten der Quellcodebasis wiederverwendet und zu verschiedenartigen Varianten zusammengesetzt [Szy97]. Die Merkmalauswahl bestimmt die Menge der Artefakte, die zur Generierung eines konkreten Produktes genutzt werden.

Gegensätzlich dazu existieren Annotationsansätze, die die merkmalspezifischen Artefakte nicht physisch trennen, sondern auf der Grundlage einer gemeinsamen Quellcodebasis arbeiten [AKL09]. Die Technik der Annotation ordnet einem Quellcodebereich ein Merkmal, unter Verwendung einer speziellen Kennzeichnung, zu. Die Generierung einer Anwendung nutzt die gesamte Basis und entfernt einen annotierten Abschnitt, wenn das an dieser Stelle wirkende Merkmal nicht ausgewählt wird [KA08a].

Der C-Präprozessor (CPP) ist eine Technik, die in die Menge der Annotationsansätze eingeordnet wird [KAK08]. Die Grundidee des auch als Makroprozessor bezeichneten Werkzeuges ist die variable Gestaltung des Anwendungs Quellcodes, um die im Fokus dieser Arbeit liegenden Anpassungen an verschiedene Plattformen vorzunehmen oder funktionelle Eigenschaften konfigurierbar zu machen [Int99, Pri93]. Die Verarbeitung stützt sich auf Makros, die zur systematischen Transformation eines Dokumentes genutzt werden [KR88]. Ein Makro erfüllt zum Einen die Aufgabe, komplexe Konstrukte einem einfachen Bezeichner zuzuordnen, um jedes Auftreten des Bezeichners mit dem gekapselten Quellcode zu ersetzen. Zum Anderen werden Makros zur Anpassung des Quellcodes verwendet und spiegeln die Zugehörigkeit zu einem Merkmal wider [Käs10]. Der CPP stellt als Annotationstechnik die Direktiven der bedingten Übersetzung bereit. Mit Hilfe dieser textuellen Markierungen, wie `#ifdef X` und `#endif`, wird ein Quellcodefragment umschlossen und eine Abhängigkeit zum Merkmal X erschaffen. Die Direktiven formulieren mit den CPP-spezifischen Sprachkonstrukten verschiedene Bedingungen. Eine konkrete SPL-Variante wird konstruiert, indem die ausgewählten Merkmale durch die Definition des dazugehörigen Makros (`#define`) bereitgestellt werden. Der CPP nutzt die getroffene Auswahl und transformiert den Quellcode durch die Auswertung der Direktiven, insbesondere der jeweiligen Bedingungen. Ein nicht gewähltes Merkmal löst das Entfernen der korrespondierenden Fragmente aus [AKL09]. Die Transformation des Quellcodes ist ein Bestandteil der Abarbeitungsphasen eines Übersetzers (engl. Compiler) der Programmiersprache C und repräsentiert eine Technik zur Implementierung einer SPL [KAK08]. Die generell weite Verbreitung in industriellen Projekten und die damit verbundene Popularität unter den Programmierern leitet sich nach FAVRE [Fav97] aus der Simplität des Werkzeuges ab. Eine überschaubare Menge an Befehlen, eine einfache Erlernbarkeit und die Flexibilität bei der konfigurationsbezogenen Anpassung einer Anwendung sind Eigenschaften, die die Popularität gefördert haben. Nach PADIOLEAU [Pad09] ist der CPP dafür verantwortlich, dass die Sprache C 35 Jahre nach der Entstehung noch immer genutzt wird und über den limitierten Funktionsumfang hinaus erweitert wurde.

Die Entwicklung einer SPL ist auf das Ziel ausgerichtet, variabel gestaltbare Anwendungen zu entwickeln. Durch den CPP wird ein bestehender und weitverbreiteter Mechanismus adaptiert und für das verfolgte Vorhaben eingesetzt [AKL09, KAK08]. Die textbasierten Transformationen werden durch die Auswertung von Direktiven in der Verarbeitung durch einen C-Compiler ausgeführt. Mit steigender Anzahl an Merkmalen nehmen die Komplexität der Anpassbarkeit und die Durchsetzung des Quellco-

des mit den Anweisungen des CPP zu. Der Anstieg wirkt sich negativ auf die Sicherstellung der Korrektheit einer SPL aus, da eine große Vielzahl an unterschiedlichen Varianten konstruiert werden kann. Auch wenn nur eine geringe Menge an Konfigurationen genutzt wird, existieren möglicherweise Millionen weitere, die anhand einer validen Merkmalauswahl generiert werden können. Die übliche Vorgehensweise des separaten Tests einer Variante ist bereits bei einer geringen Zahl an wählbaren Programmfunktionalitäten nicht praktikabel einsetzbar. Eine einzelne Sequenz aus Konfiguration, Übersetzung und Test muss bei n Merkmalen für eine exponentiell wachsende Menge (bis zu 2^n Varianten) realisiert werden [CP06, KA08b].

Zur Bewältigung dieses Problems existieren verschiedene Mechanismen, die eine gesamte SPL und somit alle Varianten gleichzeitig prüfen. Hierbei wird das Ziel verfolgt, dass unabhängig von der Nutzungsfrequenz einer Konfiguration die korrekte Funktionsweise sichergestellt ist. Der Ansatz eines produktlinienorientierten Typsystems prüft eine SPL auf die Typkorrektheit aller enthaltenen Varianten [CP06, TBKC07, Käs10]. Die Typfehler repräsentieren eine Klasse von Fehlern, die durch ein Typsystem ermittelt werden können. Ein Typfehler wird durch das Typsystem der verwendeten Programmiersprache erkannt und wird hervorgerufen, wenn beispielsweise auf eine nicht existierende Variable zugegriffen wird oder ein Funktionsaufruf „ins Leere“ führt [Car97, Pie03]. Im Hinblick auf eine SPL müssen die speziellen Prüfungen unter Beachtung der variablen Gestaltung geschehen. Die Fehler sind schwierig festzustellen, wenn sie nur vereinzelt oder bei bestimmten Merkmalkombinationen auftauchen. Ein produktlinienorientiertes Typsystem ist auf das Erkennen von dieser Art der Fehlkonfiguration ausgerichtet und ermöglicht die effiziente Überprüfung der vollständigen SPL. Diese Art der Typsysteme existieren bereits für Java und wurden detailliert untersucht [KA08b, AKGL09].

Die Programmiersprache C wird in vielen Gebieten verwendet, um Anwendungen zu entwickeln. Der CPP wird in diesem Zusammenhang für die Anpassung an funktionelle Eigenschaften oder unterschiedliche Plattformen sowie Systemen genutzt. Demnach stellt die Praxis eine große Anzahl an SPLs bereit, die für die Kombination aus C-Präprozessor und C auf die Typkorrektheit zu prüfen sind, um Fehler in der Nutzung herauszustellen. Die Entwicklung eines Analysesystems steht neuen Herausforderungen gegenüber, die durch den CPP als Technik zur Konfiguration eines Programms entstehen [Fav95]. Die gemeinsame Verarbeitung der Sprache C und der Präprozessorvariabilität stellt ein spezielles Hindernis für die Realisierung eines Untersuchungswerkzeuges bereit [Pad09].

1.2 Ziel der Arbeit

Das Ziel der vorliegenden Arbeit ist die Prüfung der Realisierbarkeit beziehungsweise die Entwicklung eines produktlinienorientierten Typsystems für C-Anwendungen, die durch den C-Präprozessor an nutzerspezifische Bedürfnisse angepasst werden. Der Schwerpunkt liegt hierbei auf der Analyse von Referenzen, wie dem Aufruf von Funktionen oder dem Zugriff auf Variablen, innerhalb der Programmiersprache C.

Zur Erfüllung dieses Vorhabens ist ein Untersuchungswerkzeug zu konzeptionieren und umzusetzen. Hierbei werden verschiedene Ziele verfolgt:

- Die Verarbeitung von Dokumenten mit der verschränkten Nutzung von Elementen der Programmiersprache C und den Direktiven des CPP.
- Der Erhalt der ursprünglichen Funktionalität des C-Präprozessors, um für alle Varianten eine korrekte Anpassung des Quellcodes sicherzustellen.
- Die Extraktion der benötigten Referenzinformationen aus einem Quellcodedokument unter Beachtung der vorhandenen Variabilitätsabhängigkeit.
- Das Aufzeigen von sprachspezifischen Referenzbeziehungen und Analyse der gesamten SPL auf Typfehler, die durch den Einsatz des CPP zur Konfiguration hervorgerufen werden.
- Das Herausstellen von Grenzen und Problemen der statischen Referenzanalyse durch die Untersuchung des Quellcodes von bestehenden Anwendungen.

1.3 Beitrag der Arbeit

Im Rahmen dieser Arbeit werden Beiträge erbracht, die im Folgenden zusammengefasst werden:

- Mit dem Ziel der statischen Referenzanalyse wird mit *TypeChef* ein prototypisches Werkzeug erschaffen, das die Untersuchung von Elementbeziehungen in einer bestehenden C-Anwendung ermöglicht. Jedes ermittelte und durch den CPP annotierte Referenzpaar wird für alle Varianten einer Anwendung auf die Verursachung eines Typfehlers geprüft.
- Im Hinblick auf das verfolgte Vorhaben wurde ein Konzept für *TypeChef* entwickelt, das die Einflussnahme des CPP auf die in Beziehung stehenden Elemente der Sprache C zu untersucht. Hierbei werden verschiedene Gesichtspunkte beachtet. Der Quellcode beinhaltet zusätzlich Direktive, die nicht an der Ausführung der Konfiguration beteiligt sind, allerdings hinsichtlich der innewohnenden Funktionalität beachtet werden müssen. Weiterhin sind die C-Referenzen unter Betrachtung der Annotation zu verarbeiten und die spezifischen Beziehungen herzustellen, um das Vorhandensein eines Typfehlers festzustellen. Das Konzept berücksichtigt die Komplexität der Bedingungen, die seitens des CPP bei der Transformation ausgewertet und in den Direktiven formuliert werden können.
- Das Konzept für die statische Referenzanalyse wurde in *TypeChef* prototypisch implementiert. Mit Hilfe des partiellen C-Präprozessors wird die Transformation eines C-Dokumentes vollzogen, ohne die darin enthaltene Variabilität aufzulösen. Der Kontext der bedingten Übersetzung bleibt vollständig erhalten und wird durch einen speziell erweiterten Parser genutzt, um die Elemente von C in Abhängigkeit zu den umschließenden CPP-Anweisungen zu analysieren und bestehende

Referenzbeziehungen herauszustellen. Auf Grund der möglichen Komplexität einer bedingten CPP-Direktive transformiert *TypeChef* die Annotationen in die Bestandteile eines Bedingungserfüllungsproblems. Dieser Mechanismus erlaubt die automatisierte Prüfung der Referenzen auf die Existenz eines Typfehlers für alle Varianten einer untersuchten Anwendung.

- Im Rahmen einer Fallstudie werden die Analyseresultate von *TypeChef* evaluiert, die während der statischen Referenzanalyse in einem bestehenden C-Projekt ermittelt werden. Aus der Entwicklung von *TypeChef* sowie der Evaluation der Ergebnisse werden Grenzen und Probleme des verfolgten Vorhabens aufgezeigt, die in zukünftigen Arbeiten näher zu untersuchen sind.

1.4 Gliederung der Arbeit

In Kapitel 2 werden die Grundlagen für den C-Präprozessor, die Softwareproduktlinien und die Typsysteme vermittelt. Das Konzept für ein produktlinienorientiertes Typsystem ist der Inhalt von Kapitel 3. Auf der Basis der dort gewonnenen Erkenntnisse beschreibt Kapitel 4 die prototypische Implementierung der einzelnen Phasen des Analysesystems. In Kapitel 5 wird die statische Referenzanalyse im Rahmen einer Fallstudie durchgeführt und die Ergebnisse evaluiert. Die hierbei auftretenden Problemfelder werden in Kapitel 6 diskutiert und Lösungen vorgeschlagen. Kapitel 7 dient der Abgrenzung zu verwandten Themengebieten. Den Abschluss der Arbeit bildet Kapitel 8, in dem die Ergebnisse zusammengefasst werden und ein Ausblick gegeben wird.

KAPITEL 2

Grundlagen

In diesem Kapitel werden die Grundlagen vermittelt, die zum Verständnis der Arbeit beitragen sollen. Der Abschnitt 2.1 beschreibt die Grundzüge des C-Präprozessors, den enthaltenen Funktionsumfang und die Arbeitsweise der im Fokus stehenden Direktiven. Anschließend wird das Prinzip von Softwareproduktlinien und die zur Umsetzung angewendeten Techniken, speziell die C-Präprozessor-gesteuerte Erzeugung von Variabilität, erläutert (Abschnitt 2.2). Der Abschnitt 2.3 schildert die Modellierung von Merkmalen und beschreibt das Merkmaldiagramm als eine grafische Repräsentation. Weiterhin wird Transformation des Diagramms in ein automatisiertes Lösungssystem erläutert, wodurch eine Merkmalauswahl auf Validität geprüft werden kann. Die Erweiterung der Darstellung ermöglicht die Integration von Zusatzinformationen, die die Aussagekraft der Grundform um zusätzliche Attribute ergänzt. Den Abschluss des Kapitels bildet eine Einführung in die Funktionsweise eines Typsystems und die Betrachtung der Programmiersprache C hinsichtlich der Eigenschaften des verwendeten Typsystems (Abschnitt 2.4).

2.1 C-Präprozessor

Als Bestandteil eines Übersetzers für die Programmiersprache C wird der CPP vor der eigentlichen Übersetzung zur textuellen Transformation des Quellcodes genutzt. Der CPP wird auch als *Makroprozessor* bezeichnet und stellt demnach eine Möglichkeit für das Anlegen von *Makros* als Abkürzung für komplexe Konstrukte bereit [SW10]. Die gekapselte Funktionalität kann mittels eines Makros verschiedenartig innerhalb des Anwendungs Quellcodes wiederverwendet werden. Die Hauptaufgaben des CPP sind das Einfügen von Dateiinhalten, die Ersetzung von Makros und die Quellcodetransformation im Rahmen der bedingten Übersetzung [Int99]. Jene Direktiven werden im Folgenden näher erläutert, da sie in C-Anwendungen zur Variabilitätserzeugung dienen und somit im Untersuchungsfokus stehen.

2.1.1 Einfügen von Dateiinhalten

Eine mit dem Schlüsselwort `#include` beginnende Quellcodezeile bewirkt das Einlesen einer Datei in das aktuelle Dokument. In Abbildung 2.1 (Seite 8) ist die Zusammensetzung der Einfügedirektive dargestellt. Die Direktive wird durch den Inhalt der Datei, die mit Hilfe von *dateiname* spezifiziert wird, ersetzt [KR88]. Der Unterschied in der Angabe des Namens beeinflusst dabei ausschließlich die Lokalisierung des einzubindenden Quellcodedokumentes [Pri93].

```

SYNTAX:
  Inklusion: #include ( <dateiname> | "dateiname" );
ANWENDUNGSBEISPIEL:
  #include <stdio.h>
  #include "project.h"

```

Abbildung 2.1: Syntax und Anwendungsbeispiel für die Dateiinklusion [KR88]

Das Beispiel in Abbildung 2.1 zeigt das Einbinden von zwei Beschreibungsdateien (engl. Header-Files) aus unterschiedlichen Suchverzeichnissen. **#include**-Direktiven können geschachtelt genutzt werden, sodass eine eingebundene Datei weitere Befehle dieses Typs enthalten kann, um eine Hierarchie für die Kapselung in funktionelle Einheiten zu erreichen [Pri93]. Die mögliche Schachtelung ist verantwortlich für das Auftreten von rekursiven Inklusionen. Durch die Nutzung eines Einfügeschutzes (engl. Include Guard) wird ein wiederholtes Ausführen der Direktive verhindert.

2.1.2 Definition von Makros und symbolischen Konstanten

Der CPP stellt mit der Direktive **#define** eine Möglichkeit bereit, Makros zu definieren. Mit Hilfe eines Makros wird einem Quellcodefragment ein *bezeichner* zugeordnet. Die Makroersetzung bewirkt eine textuelle Transformation, die die angegebene *zeichensequenz* für jedes Auftreten der Produktionsregel *bezeichner* einfügt [SW10]. Bei der Verwendung von Makros wird eine beliebige Verschachtelung ermöglicht, sodass ein Makro in einer anderen **#define**-Direktive eingesetzt werden kann, wenn es im vorhergehenden Quellcodeabschnitt erschaffen wird [Pri93].

Makro

Die Festlegung eines Makros erfolgt syntaktisch ähnlich einem Funktionsaufruf (vgl. Abbildung 2.2). Alle Elemente der Produktion *bezeichner-liste* sind innerhalb des Regelteils *zeichensequenz* für komplexere Auswertungen des Ausdrucks nutzbar [Pri93]. Das Beispiel in Abbildung 2.2 zeigt ein Makro MAX und die *bezeichner-liste*, deren Elemente im Quelltext, der an Stelle von MAX eingesetzt wird, für weitere Auswertungen verwendbar sind.

```

SYNTAX:
  Makrodef: #define bezeichner (bezeichner-liste)
            zeichensequenz;
  bezeichner-liste: bezeichner (, bezeichner)*;
ANWENDUNGSBEISPIEL:
  #define MAX(a,b) ((a) > (b)) ? (a) (b)

```

Abbildung 2.2: Syntax und Anwendungsbeispiel eine Makrodefinition [KR88]¹

¹ * - 0-n-faches Auftreten des Ausdrucks

Symbolische Konstante

Eine symbolische Konstante ist eine spezialisierte Form eines Makros. Die festgelegte Regel *bezeichner* wird durch einen konstanten Wert, hier die *zeichensequenz*, substituiert [Pri93]. Die Abbildung 2.3 zeigt eine Beispieldefinition, die jedes Vorkommen des Bezeichners `BUFF_SIZE` durch die Zeichenkette „512“ ersetzt.

```
SYNTAX:
Makrodef: #define bezeichner zeichensequenz;
ANWENDUNGSBEISPIEL:
#define BUFF_SIZE 512
```

Abbildung 2.3: Syntax und Anwendungsbeispiel einer Makrodefinition als symbolische Konstante [KR88]

2.1.3 Bedingte Übersetzung

Für die bedingte Übersetzung wird eine Menge von Direktiven zur Verfügung gestellt: `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else` und `#endif`. Eine Kombination der Befehle versetzt den Programmierer in die Lage, den Quellcode, in Abhängigkeit von einer festgelegten Konfiguration, anpassen zu lassen. Demnach werden bei der bedingten Übersetzung alle Quellcodeabschnitte aus der Datei entfernt, die nicht Bestandteil der aktuellen Konfiguration sind [Wol09].

Die konstanten Ausdrücke in den `#if`- und den beliebig vielen, nachfolgenden `#elif`-Zeilen werden der Reihe nach bewertet, bis eine Bedingung einen von 0 verschiedenen Wert liefert. Der zu dieser Zeile gehörende Quellcodeabschnitt bleibt im Dokument enthalten. Sobald keiner der Ausdrücke zutrifft, wird die `#else`-Direktive ausgeführt. Ist eine solche Zeile nicht vorhanden, so werden alle umschlossenen Blöcke nicht bearbeitet [Wol09].

Zum Beispiel wird der Ausdruck `foo(1)` in Abbildung 2.4 (Seite 10) erreicht, falls `BUFF_SIZE` den Wert 0 annimmt, beziehungsweise der Abschnitt `foo(2)`, wenn das Makro mit dem Wert 512 definiert wurde. Treffen die beiden Bedingungen nicht zu, wird in den `#else`-Fall verzweigt und nur die Anweisung `foo(3)` bleibt Bestandteil der Quelldatei.

Im Rahmen der Erzeugung von Variabilität werden die Direktiven `#if` und `#elif` zum Auswerten von Gleichungen zu einem konstanten Ausdruck genutzt [SW10]. Eine Gleichung kann neben den expandierten Makros zusätzlich Ganzzahl- und Symbolkonstanten beinhalten. Für die Verknüpfung der Bestandteile sind folgende Operatoren zu verwenden:

arithmetisch: Addition, Subtraktion, Multiplikation, Division, Modulo

bitweise: und-Verknüpfung, oder-Verknüpfung, exklusiv-oder-Verknüpfung, Komplement, Rechtsverschiebung, Linksverschiebung

logisch/ relational: größer(-gleich), kleiner(-gleich), (un-)gleich, und-Verknüpfung, oder-Verknüpfung

```

SYNTAX:
Konditional: if-teil (elif-teil)* (else-teil)? #endif;
if-teil:      #if konstanter-ausdruck c-code;
              #ifdef bezeichner c-code;
              #ifndef bezeichner c-code;
elif-teil:    #elif konstanter-ausdruck c-code;
else-teil:    #else c-code;
konstanter-ausdruck: operand (operator openrand)* ;
operand: (bezeichner | symbol | ganzzahl);
operator: ( + | - | * | / | % | & | | | ^ | ~ | >> | <<
           | > | >= | < | <= | == | != | && | || );
ANWENDUNGSBEISPIEL:
#if BUFF_SIZE == 0
    foo(1)
#elif BUFF_SIZE == 512
    foo(2)
#else
    foo(3)
#endif

```

Abbildung 2.4: Syntax und Anwendungsbeispiel für die bedingte Übersetzung [KR88]
23

Im Zuge der Auswertung eines Ausdruckes werden die einzelnen Bestandteile berechnet beziehungsweise ermittelt und wiederum ein konstanter Wert gebildet.

Ein spezieller Operator ist das Schlüsselwort (!)defined, der mit #if und #elif kombinierbar ist. Die Auswertung der Direktive ermittelt, ob ein Makro im vorhergehenden Quellcode definiert beziehungsweise nicht definiert wird. Ausgehend vom Ergebnis der Bedingungsauwertung führt der CPP eine Transformation des umrahmten Quelltextes aus [KR88]. Das Äquivalent zu der Kombination mit defined wird durch die in Abbildung 2.5 aufgeführten Direktiven #ifdef und #ifndef dargestellt. Die beiden Schreibweisen realisieren die gleiche Funktionalität. Das enthaltene Beispiel zeigt eine Annotation in Abhängigkeit von der Definition des Makros BUFF_SIZE (vgl. Abbildung 2.5).

```

SYNTAX:
#ifdef bezeichner (äquivalent: #if defined bezeichner)
#ifndef bezeichner (äquivalent: #if !defined bezeichner)
ANWENDUNGSBEISPIEL:
#ifdef BUFF_SIZE (äquivalent: #if defined BUFF_SIZE)

```

Abbildung 2.5: Spezialformen der bedingten Übersetzung [KR88]

2.1.4 Weitere Direktiven

Der Befehlssatz des CPP beinhaltet zusätzlich, zu den bisher erläuterten, weitere Direktiven, die nur tabellarisch im Zusammenhang mit ihren Funktionen aufgeführt werden, da sich die hervorgerufenen Änderungen nicht auf die in der vorliegenden Diplomarbeit angestellten Untersuchungen auswirken (vgl. Tabelle 2.1, S. 11).

² ? - 0-1-faches Auftreten des Ausdruckes

³ * - 0-n-faches Auftreten des Ausdruckes

Direktive	Funktion
#undef	Bewirkt das Entfernen der Definition eines Makros
#line	Manipulation von Zeilennummern für Querverweise und Fehlermeldungen
#pragma	Festlegung von Compiler-abhängigen Befehlen
#error	Ausgabe von Fehlermeldungen und Abbruch der Übersetzung
#	leere CPP-Anweisung

Tabelle 2.1: Weitere Direktiven des CPP

2.2 Softwareproduktlinien

In dem Bereich der Entwicklung einer Software finden unterschiedliche Strategien Anwendung, um eine Anpassung an einen bestimmten Nutzerkreis zu erreichen. Nach POHL et al. [PBL05] erfolgt die Abgrenzung bezüglich des gegensätzlichen Fokus, wodurch folgende Unterteilung vorgenommen wird:

Individualsoftware: Ausgehend von der Ausrichtung auf spezielle Anforderungen wird die Software eigens für eine gewisse Nutzergruppe realisiert. Für eine sich im Anwendungsgebiet unterscheidende Nutzergruppe wird wiederum eine maßgeschneiderte Neuentwicklung benötigt. Daraus resultieren höhere Kosten und ein größerer Aufwand bei der Entwicklung sowie ein verkleinertes Spektrum hinsichtlich der Wiederverwendung.

Massensoftware: Eine Massensoftware stellt eine breitere, allerdings allgemeinere Funktionalitätsvielfalt bereit und ist für einen großen Anwenderbereich konzipiert. Im Vergleich zu einer Individualsoftware ist die Entwicklung unter Verwendung dieser Strategievariante als kostengünstiger anzusehen.

Durch die Kombination der beiden Strategien wird die Entwicklung einer Anwendung in Form einer SPL erschaffen. Hierbei wird eine große Produktpalette bereitgestellt, ohne jedes Produkt individuell neu zu entwickeln. Eine Basis, in der alle Gemeinsamkeiten definiert und implementiert sind, ist der Ausgangspunkt, um die Anwendungen auf die Bedürfnisse der speziellen Anwender zuzuschneiden. CLEMENTS und NORTHROP legen für eine SPL folgende Definition fest:

„A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“ [CN01]

Eine SPL beschreibt folglich eine Menge von Anwendungen, die bestimmte *Merkmale* gemeinsam haben. Ein Merkmal repräsentiert eine Abstraktionsform für Programmcharakteristiken, um Gemeinsamkeiten und Unterschiede der Varianten einer SPL zu definieren. Alle Anwendungen einer SPL sind für ein bestimmtes Marktsegment vorgesehen und genügen den Anforderungen einer festgelegten Domäne. Eine Domäne

ist als „ein spezieller Wissensbereich, ein bestimmtes Fachgebiet oder eine Sammlung entsprechender Funktionalität“ charakterisiert [CN01].

Nach der Definition von CLEMENTS und NORTHROP bildet eine Menge von Artefakten die Grundlage, auf der die Entwicklung einer SPL durchgeführt wird. Ein Artefakt repräsentiert hierbei einen wiederverwendbaren Bestandteil der Software [Wit96]. Die Bestandteile werden folgendermaßen klassifiziert [CN01]:

Kernartefakt: Ein Kernartefakt stellt eine Gemeinsamkeit aller aus einer SPL generierbaren Produkte dar. Die Menge der Kernartefakte beschreibt die Basis für die Entwicklung und ist in jeder hervorgehenden Anwendung enthalten.

Produktspezifisches Artefakt: Im Gegensatz zu einem Kernartefakt wird durch ein produktspezifisches Artefakt eine speziell angepasste Funktionalität festgelegt. Die Anpassung ist ausschließlich in einem individualisierten Produkt der SPL enthalten.

Eine Menge von ausgewählten Artefakten wird unabhängig von dem jeweiligen Typ zusammengesetzt. Aus diesem Prozess geht ein konkretes Produkt der SPL hervor, das auf die entsprechenden Nutzeranforderungen zugeschnitten ist.

Der Produktlinienansatz vereint die im Bereich der Anwendungsentwicklung verbreiteten Strategien für Individual- und Massensoftware mit dem Ziel, von den jeweiligen Vorteilen zu profitieren. POHL et al. [PBL05] stellen die Entwicklungskosten von Einzellösungen und die Produkte einer SPL gegenüber: Ab ungefähr drei Produkten, die auf der Grundlage einer SPL realisiert werden, ergeben sich verschiedene Vorteile aus der Nutzung dieser Strategie. Die Entwicklungszeit und -kosten werden gesenkt, da die Kernartefakte vorhanden sind und eine Neuentwicklung für ein weiteres Produkt nicht durchgeführt werden muss [Wit96, CN01, Ape07]. Die in der Basis enthaltenen Artefakte werden bei jeder Generierung wiederverwendet, fortlaufend überprüft und getestet. Weitere Fehler werden identifiziert und behoben, sodass eine Qualitätsverbesserung für die gesamten SPL resultiert [CN01, PBL05]. Zusätzlich wird der spätere Wartungsaufwand für die Produkte reduziert, da die Änderungen an Kernartefakten eine Korrektur aller anderen Produkte hervorruft [CN01].

2.2.1 Domänen- und Anwendungsentwicklung

Die Entwicklung einer SPL verläuft in zwei parallel ablaufenden Entwicklungsprozessen. Die Domänenentwicklung (engl. Domain-Engineering) und die Anwendungsentwicklung (engl. Application-Engineering) bezeichnen die Vorgänge, die der Erstellung einer SPL zugrunde liegen und die Generierung von konkreten Produkten bewirken [CE00]. Im Gegensatz zu der konventionellen Entwicklung von Einzellösungen, bei der ausschließlich die Anforderungen eines speziellen Nutzerkreises betrachtet werden, befindet sich bei der Domänenentwicklung die vollständige Domäne im Fokus der Teilprozesse Analyse, Design und Implementierung [Cza05]. Hierbei werden neben den aktuellen auch alle potenziellen Nutzer und speziell deren Anforderungen berücksichtigt. Im Zuge der Domänenentwicklung werden die Gemeinsamkeiten und Unterschiede herausgestellt und abschließend implementiert, um in verschiedenartiger Kombination in den Varianten der SPL wiederverwendet zu werden [CE99, FKW04].

Die Anwendungsentwicklung bezeichnet den Vorgang zur Erzeugung von Varianten einer SPL. In den Teilprozessen werden die Ergebnisse der Domänenentwicklung genutzt, aber auch rückwirkend angepasst [Cza05]. Das Zusammensetzen der implementierten Kernartefakte führt zu konkreten Varianten, die hinsichtlich der Nutzerbedürfnisse individualisiert sind, da die Merkmale, gemäß der herausgestellten Anforderungen, kombiniert werden [CE00].

Die Abbildung 2.6 zeigt die beiden Prozesse, die jeweils durchgeführten Phasen der Entwicklung sowie die auftretenden Informationsflüsse. Vorrangig wird mit der Domänenentwicklung das Ziel verfolgt, die Domäne zu untersuchen, zu definieren und abschließend die implementierten Kernartefakte zu erhalten. Aus den Kernartefakten werden daraufhin in der Anwendungsentwicklung konkrete Varianten erstellt. Somit laufen Implementierung und Generierung in getrennten Phasen ab. Eine spezielle Funktionalität, die nicht für die ganze Domäne gilt, wird als neue Anforderung festgelegt und in einem gesonderten Anpassungsvorgang in Form eines produktspezifischen Artefaktes bewerkstelligt (vgl. Abbildung 2.6).

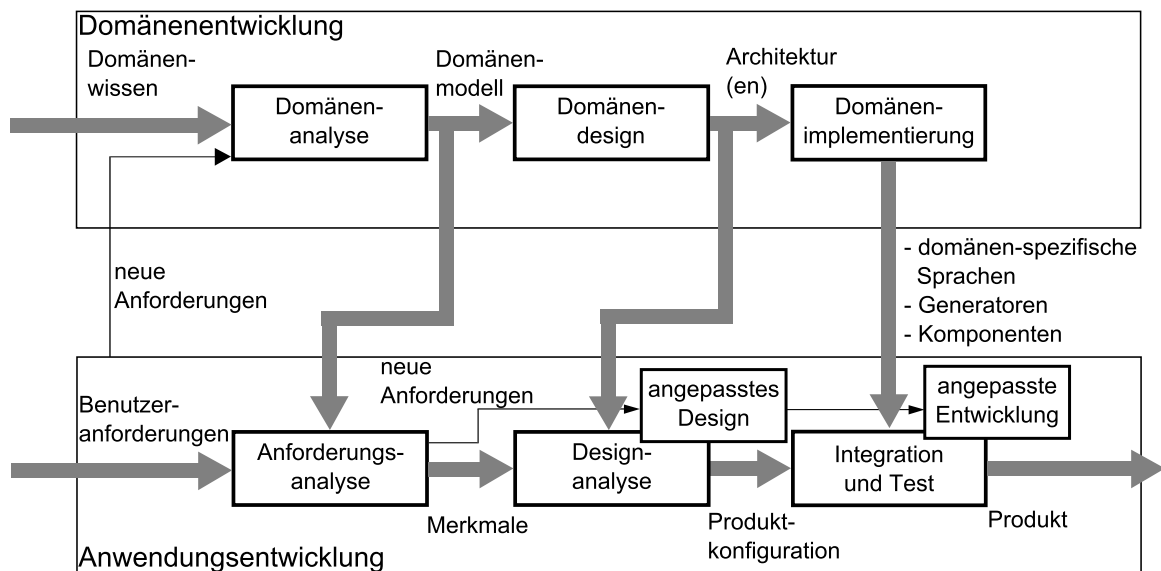


Abbildung 2.6: Domänen- und Anwendungsentwicklung [CE00]

2.2.2 Softwareproduktlinienimplementierung

Für die Implementierung einer SPL steht eine Vielzahl von Methoden, Sprachen und Werkzeugen zur Auswahl [AKL09]. Nach KÄSTNER et al. [KAK08] können die verfügbaren Techniken in zwei Klassen von Ansätzen unterteilt werden. Dabei werden die Klassen Komposition und Annotation gegenübergestellt.

Im Kompositionsansatz wird generell das Prinzip der *Trennung von Belangen* (engl. *Separation of Concerns (SoC)*) verfolgt, wodurch die Modularisierung einer Anwendung hervorgerufen wird [Dij76, Par76, Par79]. Die Belange, hier die Merkmale einer SPL, werden als eigenständige und physisch voneinander getrennte Quellcodeeinheiten

umgesetzt. Ausgehend von der Quellcodebasis werden die Programmvarianten generiert. Dieser Vorgang wird als Komposition bezeichnet und beschreibt die Wiederverwendung von vorgefertigten Einheiten in neuen Kombinationen [Szy97]. Bei der Realisierung des Ansatzes werden verschiedene Techniken angewendet [AKL09, KAK08]. Neben der objektorientierten Programmierung, bei der eine Abstraktion in Form von Klassen genutzt wird, können beispielsweise Frameworks oder Komponenten verwendet werden [JF88, Szy97, BCS00]. Bei dieser Art von Modularisierung treten Probleme auf, die eine vollständige Trennung der Merkmale in eigene Artefakte verhindern [OT01]. Mit Hilfe neuer Paradigmen, wie der subjekt-, der aspekt- oder der merkmalsorientierten Programmierung werden Techniken, die diesen Problemen entgegenwirken, bereitgestellt [HO93, LKL⁺97, Bat03].

In der anderen Klasse von Implementierungsansätzen kommen Annotationen zum Einsatz, um die Belange voneinander zu trennen [AKL09]. Hierbei wird die Trennung virtuell und somit, gegensätzlich zum Kompositionsansatz, nicht physisch durchgeführt. Die Merkmale sind nicht in abgeschlossenen Einheiten modularisiert, sondern werden innerhalb der vollständig vorliegenden Quellcodebasis zugeordnet. Bei der Generierung einer Variante werden alle nicht ausgewählten Merkmale beziehungsweise der dazugehörige Quellcode aus der Basis entfernt. KÄSTNER et al. [KTA08] führt verschiedene Techniken auf, die den Annotationsansatz verfolgen. Dazu gehören neben der bedingten Übersetzung durch einen Präprozessor auch Frames/XVCL [JBZZ03], Explicit Programming [BCDVM02] oder Software Plans [CPR07].

In diesem Zusammenhang wird der C-Präprozessor als eine Möglichkeit vorgestellt, die Merkmale innerhalb des Quellcodes textbasiert zu annotieren [KTA08, KA09]. Nach FAVRE [Fav96] ist der CPP ein in der Industrie und der Forschung weitverbreitetes Werkzeug. Der bereitgestellte Befehlssatz, speziell die Direktiven der bedingten Übersetzung, bildet die Grundlage für die Erzeugung von Variabilität [MP03, ADMTH09]. An dieser Stelle wird ein Quellcodeabschnitt, der nur in bestimmten Varianten einer Anwendung enthalten sein soll, beispielsweise mit den Direktiven `#ifdef` und `#endif` umschlossen und gemäß der Merkmalauswahl verarbeitet (vgl. Abschnitt 2.1). Die Auswahl erfolgt durch eine parametrisierte Übersetzung des Projektes oder die Definition von Makros innerhalb der im Projekt enthaltenen Dateien. Die Konfiguration beeinflusst die Quellcodetransformationen, die der CPP während des Übersetzungsprozesses in einem separaten Verarbeitungsschritt bewerkstelligt.

Kritik am CPP

Der Einsatz des CPP steht jedoch in der Kritik, da die Anwesenheit von CPP-Direktiven im Quellcode nachteilig sein kann und verschiedene Probleme hervorruft [SC92]. Die Kritik stützt sich nach KÄSTNER und APEL [KA09] auf die folgenden Problemfelder:

Mangelhafte Trennung von Belangen: Zwei der größten Probleme des CPP sind die Modularität und die Nachvollziehbarkeit. Der merkmalspezifische Quellcode wird nicht in eine separate Einheit gekapselt, sondern ist über alle Dateien verstreut und innerhalb einer Datei mit anderen Merkmalen verwoben. Nach FAVRE [Fav95, Fav97] muss für das Verständnis eines Codefragmentes meist Wissen über das gesamte Projekt vorliegen. Ebenso werden Anpassungs- und

Wartungsarbeiten erschwert und müssen mit erhöhtem Aufwand durchgeführt werden, da eine CPP-Anweisung nicht immer den Rückschluss auf das an dieser Stelle greifende Merkmal zulässt [Fav97]. Speziell das Entfernen einer Konfigurationsvariable und des dazugehörigen Quellcodes kann eine langwierige Prozedur sein, wenn sie manuell realisiert wird [Fav96, BM01].

Fehleranfälligkeit: Die Benutzung der Direktiven des CPP zur Annotation von Merkmalen im Quellcode ist anfällig für verschiedene Arten von Fehlern. Eine mögliche Klasse sind Syntaxfehler [KAT⁺09]. Der CPP ist ein zeichenbasiertes Werkzeug, das für den Quellcode keine Interpretation durchführt. Das Erkennen eines solchen Fehlers kann durch einen Compiler geprüft werden, ist allerdings abhängig von der Variantenanzahl. Sobald jede Möglichkeit einzeln konfiguriert und getestet wird, wächst parallel zu der Anzahl der Varianten auch der Aufwand für die Fehlersuche [ADPB02]. Weiterhin können Typfehler und fehlerhaftes Programmverhalten auftreten [KAT⁺09, KAS10]. Ein Typfehler entsteht, wenn z. B. der Aufruf einer Funktion in jeder Konfiguration enthalten ist, obwohl diese nicht in allen definiert wird. Mit steigender Nutzung und der daraus resultierenden Verschachtelung von Direktiven nimmt darüber hinaus die Fehleranfälligkeit zu [SC92].

Erschwerte Quellcodewahrnehmung: Der CPP ist ein eigenständiges Werkzeug und nicht an eine Zielsprache gebunden. Demnach werden die Präprozessor Direktiven mit dem Inhalt der Datei kombiniert und vermischt. Die Sprachmischung ist ein Hauptproblem für das Verständnis eines Programms [Fav97, VB03]. Einen weiteren Einfluss auf die Quellcodewahrnehmung hat die zulässige Granularität. Ein feingranularer Einsatz des CPP, wie die konfigurationsabhängige Manipulation von Steuerstrukturen oder die variable Gestaltung des Rückgabetyps einer Funktion, führt zu einer starken Durchsetzung des C-Quellcodes [SC92]. Nach FAVRE [Fav97] können zwischen zusammengehörenden Direktiven, wie `#ifdef` und `#endif`, mehrere hundert Quellcodezeilen liegen, wodurch eine Zusammengehörigkeit nicht ersichtlich ist. Die Quellcodebereiche sind schwierig in die Konfigurationsstruktur einzuordnen, wenn feingranulare und tief verschachtelte Annotationen verwendet werden [KS94].

Erschwerte Wiederverwendung: Die Nutzung des CPP schränkt die Wiederverwendung des Quellcodes sehr stark ein. Das gezielte Herauslösen wird durch die Streuung der merkmalspezifischen Bereiche in der Basis und den speziellen Kontext, in dem diese innerhalb der aktuellen SPL genutzt werden, beeinträchtigt [KA09].

Vorteile des CPP

Die Verwendung von Präprozessoren ist jedoch nicht ausschließlich negativ behaftet. Nach KÄSTNER et al. [KAS10] können verschiedene Vorteile, die aus der Anwendung solcher Werkzeuge resultieren, herausgestellt werden:

Einfaches Programmiermodell: Der merkmalspezifische Quellcode kann annotiert und entfernt werden. Aus der überschaubaren Menge an Befehlen ergibt sich eine einfache Benutzbarkeit [Fav97, KAS10].

Sprachunabhängigkeit: Der Einsatz eines Präprozessors ist meist nicht auf eine bestimmte Zielsprache begrenzt [KA09]. Folglich werden die Transformationen des CPP auch in Dateien, die keinen C-Quellcode beinhalten, durchgeführt [Fav95]. In Anbetracht der Sprachunabhängigkeit muss die Anwendung der Direktiven nicht an die zu konfigurierende Sprache angepasst werden [KAS10].

Einfache (Dominante) Dekomposition: Die traditionelle SoC und somit die Dekomposition in Module oder Klassen ist in Verbindung mit dem CPP weiterhin möglich [KA09]. Nach KÄSTNER et al. [KAS10] fügt die Verwendung des Präprozessors eine weitere Ausdrucksfähigkeit hinzu, die mit anderen Ansätzen nicht erreicht werden kann.

2.3 Merkmalmodellierung

Im Zuge der Domänenanalyse werden die Aktivitäten der Domäneneingrenzung und die Domänenmodellierung durchgeführt. Das Resultat der zweiten Phase ist ein Modell der Domäne. CZARNECKI et al. [CE00] stellen diesbezüglich verschiedene Analysemethoden vor. Eine spezielle Methode ist die merkmalsorientierte Domänenanalyse, die das Merkmal als Beschreibungselemente benutzt [KCH⁺90, KLD02]. Dementsprechend resultiert aus dem Analysevorgang ein Merkmalmodell, das die wiederverwendbaren Anforderungen für die Domäne hierarchisch definiert und die Gemeinsamkeiten und Unterschiede der Programme der SPL beschreibt [CE00]. Das Modell repräsentiert alle Konfigurationen, die durch die Auswahl der Merkmale bezüglich ihrer Semantik bestimmt werden [CW07]. Hierbei werden Abhängigkeiten zwischen den Merkmalen beschrieben, die die Menge der validen Varianten eingrenzen. Durch die Festlegung solcher Beziehungen entstehen Informationen, die in die Analyse aller Konfigurationen einer Anwendung einbezogen werden können.

Das *Merkmalendiagramm* ist eine grafische Repräsentation, die das Verständnis von Merkmalmodellen fördert [KCH⁺90]. Eine Menge von Knoten und gerichteten Kanten definiert die Grundbestandteile des Diagramms und erschafft eine Baumstruktur, in der die Merkmale und deren Abhängigkeiten hierarchisch abgebildet werden. In der Wurzel ist die Basisimplementierung der SPL hinterlegt. Die darunter angeordneten Knoten beinhalten die während der Analyse ermittelten Merkmale. KANG et al. [KCH⁺90] beschreiben unterschiedliche Kantentypen, die eine optionale, verbindliche oder alternative Knotenbeziehung festlegen. Die Menge von Beziehungen wird durch CZARNECKI [Cza98] um eine und-Beziehung sowie eine oder-Beziehung ergänzt.

In Abbildung 2.7 (S. 17) werden die Beziehungen mit ihrer Notation im Merkmalendiagramm dargestellt. Für die Kantentypen und somit die Abhängigkeiten zwischen den Merkmalen werden folgende Eigenschaften definiert:

verbindlich: Ein verbindliches Merkmal muss ausgewählt werden, sobald der Elternknoten ein Bestandteil der Variante ist.

optional: Wird der Elternknoten gewählt, so ist dieses Merkmal optional auszuwählen.

alternativ: Alternative Beziehungen zwischen Merkmalen bewirken bei der Wahl eines Knoten den Ausschluss aller anderen Mitglieder der Verknüpfung.

und: Eine mit der und-Beziehung verbundene Knotenmenge ist vollständig zu selektieren, wenn der darüberliegende Knoten enthalten ist.

oder: Jeder verbindliche Knoten einer mit der oder-Beziehung verknüpften Menge muss gewählt werden, die optionale Eigenschaft der Kindknoten bleibt dabei erhalten.

impliziert: Die Implikation zeigt die Abhängigkeit zwischen Merkmalen an, sodass die Auswahl eines der verknüpften Merkmale auch die Auswahl des anderen impliziert.

exkludiert: Zwei mit der Exklusion verbundene Knoten schließen sich gegenseitig aus. Die repräsentierten Merkmale sind somit nicht gleichzeitig in einer Variante enthalten.

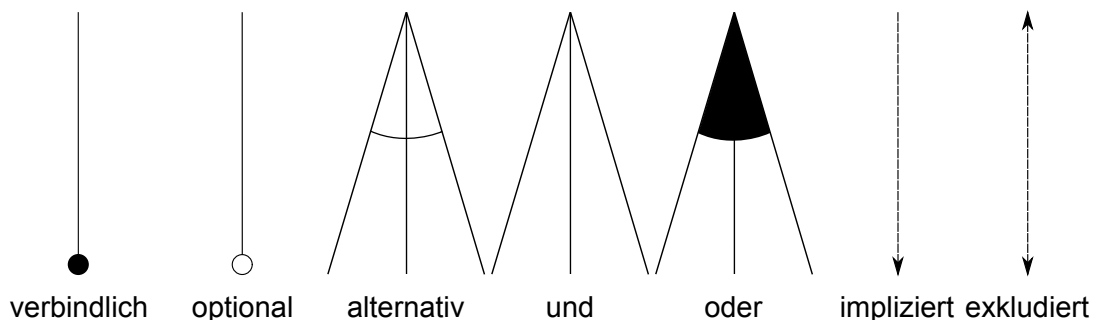


Abbildung 2.7: Notation der Beziehungen in Merkmaldiagrammen [Bat05]

Das Merkmaldiagramm ist die grafische Darstellung aller Merkmalabhängigkeiten einer SPL. Auf der Grundlage des Diagramms kann für eine Menge von gewählten Merkmalen die Validität ermittelt werden, indem die Baumstruktur in ein Format überführt wird, welches eine automatisierte Prüfung ermöglicht.

2.3.1 Transformation von Merkmaldiagrammen

Die im Merkmaldiagramm dargestellten Inhalte und modellierten Beziehungen lassen sich in eine *aussagenlogische Darstellung* transformieren [Bat05]. Ein aussagenlogischer Ausdruck ist eine Menge von Variablen, die durch logische Operatoren (\neg , \vee , \wedge , \rightarrow , \leftrightarrow) verknüpft werden. Jedes Merkmal wird als eine boolesche Variable interpretiert und nimmt den Wert wahr an, wenn das Merkmal ausgewählt wird. Der Wert der Variable wird auf falsch festgelegt, sobald das Merkmal nicht enthalten ist.

Alle enthaltenen Abhängigkeiten werden nach den Regeln aus Tabelle 2.2 (Spalte SAT-Mapping) in einen aussagenlogischen Ausdruck überführt [BSRC10]. Dieser Ausdruck wird durch ein Lösungssystem (engl. Solver) auf die Erfüllbarkeit (Boolean SATisfiability Problem (SAT)) des darin definierten Problems geprüft. Das Ergebnis der Untersuchung bestimmt, inwiefern die ausgewählte Merkmalmenge eine valide Zusammenstellung ist und somit durch das Lösungssystem mit wahr oder falsch bewertet wird [WSB⁺08]. Die Ermittlung der Validität stellt die Korrektheit der untersuchten Variante der im Merkmaldiagramm repräsentierten SPL heraus [CW07].



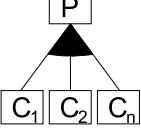
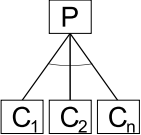
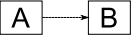

Beziehung	SAT Mapping	CSP Mapping
	$P \leftrightarrow C$	$P = C$
	$C \rightarrow P$	$\text{if}(P = 0)$ $C = 0$
	$P \leftrightarrow (C_1 \vee C_2 \vee \dots \vee C_n)$	$\text{if}(P > 0)$ $\text{Sum}(C_1, C_2, \dots, C_n) \text{ in } \{1..n\}$ else $C_1 = 0, C_2 = 0, \dots, C_n = 0$
	$(C_1 \leftrightarrow (\neg C_2 \wedge \dots \wedge \neg C_n \wedge P)) \wedge$ $(C_2 \leftrightarrow (\neg C_1 \wedge \dots \wedge \neg C_n \wedge P)) \wedge$ $(C_n \leftrightarrow (\neg C_1 \wedge \dots \wedge \neg C_{n-1} \wedge P))$	$\text{if}(P > 0)$ $\text{Sum}(C_1, C_2, \dots, C_n) \text{ in } \{1..1\}$ else $C_1 = 0, C_2 = 0, \dots, C_n = 0$
	$A \rightarrow B$	$\text{if}(A > 0)$ $B > 0$
	$\neg(A \wedge B)$	$\text{if}(A > 0)$ $B = 0$

Tabelle 2.2: Transformationsregeln für Merkmaldiagramme in aussagenlogische Ausdrücke und CSP Ausdrücke (adaptiert aus [BSRC10])

Eine weitere Möglichkeit der Analyse und Bewertung einer Auswahl ist die Interpretation der im Diagramm hinterlegten Zusammenhänge als ein System von Variablen und den dazwischen definierten Beziehungen. In Tabelle 2.2 (Spalte CSP-Mapping) ist ein weiterer Regelsatz dargestellt, der die Transformation des Diagramms in eine Bedingungsformel ermöglicht. Die vollständige Überführung konstruiert für die SPL ein *Bedingungserfüllungsproblem* (engl. Constraint Satisfaction Problem (CSP)), das wiederum zur Prüfung der Korrektheit einer Variante nutzbar ist.

Bedingungserfüllungsproblem

Ein CSP beinhaltet für die zu untersuchende Problemstellung eine Menge von Variablen [FDK93]. Für jede der Variablen wird eine finite Domäne definiert, die den Bereich der annehmbaren Werte festlegt [BSMAC06]. Des Weiteren werden Bedingungen ermittelt, die die Zusammenhänge zwischen den Variablen aufzeigen [BSTC07]. Für das Problem wird mittels eines Lösungssystems untersucht, inwiefern die Bedingungen unter den gegebenen Voraussetzungen erfüllt werden können und welche Kombinationen von Variablen, speziell deren Belegungen, bei der Erfüllung beteiligt sind.

Die von BENAVIDES et al. [BSRC10] erläuterten Regeln formen ein bestehendes Merkmaldiagramm in ein Bedingungserfüllungsproblem um. Jedes Merkmal einer SPL wird als eine Variable des CSP-Modells interpretiert und kann die Werte 0 oder 1 annehmen. Weiterhin werden alle im Diagramm dargestellten Abhängigkeiten in Bedingungen zwischen den Variablen transformiert.

Ein CSP-Solver ist ein automatisiertes Lösungssystem, welches für ein abgegrenztes Problem prüft, inwiefern eine zusammengestellte Variante korrekt und somit zulässig ist. Die dabei ausgewählten Merkmale werden durch eine Variablenbelegung ausgedrückt. Unter Verwendung der Belegung prüft der CSP-Solver das Eintreten der formulierten Bedingung.

2.3.2 Erweiterung von Merkmaldiagrammen

Nach BENAVIDES et al. [BMAC05] reicht die Aussagekraft eines Merkmaldiagramms nicht für die Darstellung von speziellen Charakteristiken einer SPL aus. Solche Eigenschaften werden als extra-funktionale Merkmale bezeichnet und ordnen jedem Merkmal eine Attributrelation zu. In der Relation werden verschiedene Attribute, die eine messbare Charakteristik, wie die Entwicklungskosten eines Merkmals, darstellen, in Beziehung gebracht. Die zusätzlichen Informationen erschaffen die grafische Darstellung, die als *erweitertes Merkmaldiagramm* bezeichnet wird.

In der Abbildung 2.8 (S. 20) ist ein Beispiel für die erweiterte Form eines Merkmaldiagramms dargestellt [BSRC10]. Jedem Merkmalknoten werden Attribute zugeordnet, die detaillierte Informationen über eine Charakteristik bereitstellen. Im Beispiel wird das Diagramm um die Geschwindigkeit, die Kosten und der vorhandene Speicher als Attribute erweitert. Hierbei werden durch die extra-funktionalen Informationen neue Bedingungen formuliert, deren Erfüllung in Hinblick auf die Auswahl des Merkmals betrachtet werden muss.

Die erweiterte Version, wie auch die Grundform, kann in ein CSP überführt werden, um an dieser Stelle Analysen und Optimierungen der Attribute durchzuführen [BMAC05]. Für die Überprüfung einer Variante auf eine korrekte Merkmalauswahl wird das Diagramm durch festgelegte Regeln transformiert. Die verwendeten Attribute werden in das CSP aufgenommen. Ein extra-funktionales Merkmal wird je nach Verwendung als reelle oder ganze Zahl in dem dazugehörigen Wertebereich interpretiert. In Abbildung 2.8 (S. 20) werden reelle Werte vereinbart. Der Wertebereich kann im Diagramm festgelegt sein oder wird durch das verwendete Lösungssystem vorgegeben. In die bisherigen Überprüfungen des CSP-Solvers fließen die Variablen und Bedingungen aus dem Kontext der Attribute ein und werden zum Herausstellen einer korrekten

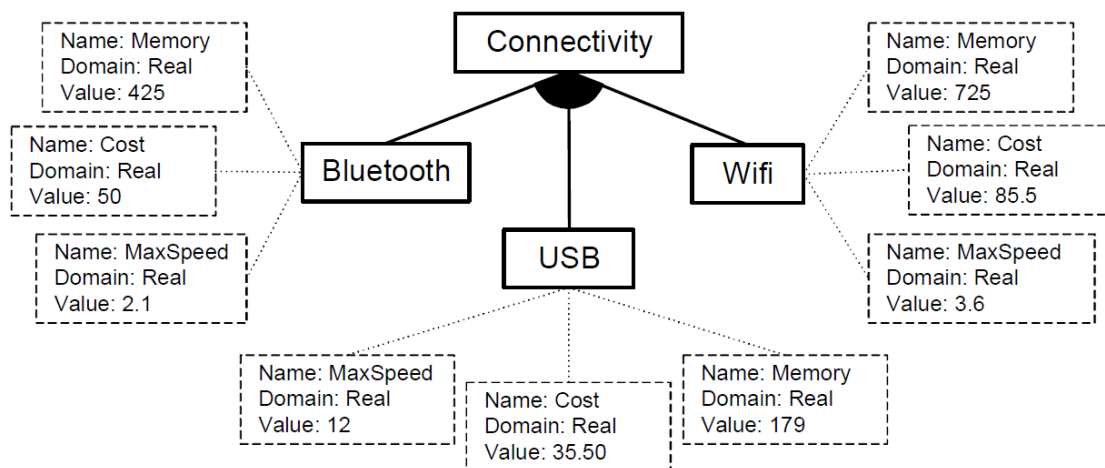


Abbildung 2.8: Erweitertes Merkmaldiagramm [BSRC10]

Merkmalselektion verwendet.

Die zusätzlichen Attribute eines Merkmaldiagramms besitzen eine ähnliche Mächtigkeit wie der CPP. Die Art der Bedingungen, die hinsichtlich der Attribute formuliert wird, ist mit den Anweisungen der bedingten Übersetzung vergleichbar. Die darin hinterlegten Ausdrücke werten ebenso Variablen, hierbei als Repräsentation eines Makros, in verschiedenartigen Bedingungen aus. Auf dieser Vorgehensweise wird aufgebaut, um die Variabilität des CPP als Bedingungserfüllungsproblem zu untersuchen.

2.4 Typsysteme

In der modernen Softwareentwicklung existiert eine Vielzahl von formalen Methoden, die sicherstellen sollen, ob ein System ein korrektes Verhalten aufzeigt [Car97]. Die Untersuchungen stützen sich dabei auf implizite oder explizite Spezifikationen des zu erwartenden beziehungsweise des gewünschten Verhaltens. Das Spektrum der formalen Methoden reicht von Frameworks, wie Hoare Logiken oder algebraische Spezifikationssprachen, bis hin zu automatischen Überprüfungsverfahren, die Bestandteil von Compilern oder aber Analysesystemen sind. Die bekannteste und am meisten verbreitete Methode ist das *Typsystem*, das nach PIERCE [Pie02] folgendermaßen definiert wird:

„A type system is a traceable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.“ [Pie02]

Durch ein Typsystem können folglich Schlussfolgerungen über ein Programm getroffen werden, indem auf das Nichtvorhandensein eines spezifizierten Programmfehlverhaltens geprüft wird, um das Auftreten von Ausführungsfehlern während der Laufzeit zu verhindern. Ein solches Fehlverhalten wird auch als Typfehler bezeichnet und kann mit Hilfe der verwendeten Methode eliminiert werden [Car97].

Grundsätzlich werden die Programmiersprachen in *getypt* und *ungetypt* unterteilt. Eine Sprache ist *getypt*, sobald den verwendeten Variablen ein Typ, speziell ein fester Wertebereich, zugeordnet werden kann. Beispielsweise wird für eine Variable vom Typ Boolean erwartet, dass sie in jedem Programmdurchlauf ausschließlich logische Werte annimmt. Die Typüberprüfungen der im Programm verwendeten Datentypen werden *statisch* zur Übersetzungszeit durchgeführt. In einer ungetypten Sprache existieren für die Variablen keine Restriktionen hinsichtlich des Wertebereiches, sodass die Programmiersprache keinen Typen oder ausschließlich eine universelle Form nutzt. Die Untersuchung auf ein mögliches Fehlverhalten wird dynamisch zur Laufzeit realisiert [Car97].

Im Zusammenhang mit der Typisierung einer Sprache existieren zwei Prinzipien, die die Art der Unterscheidung von Typen beschreibt. Die Typisierung ist als stark/streng anzusehen, wenn alle unzulässigen Operationen auf Datentypen, wie z. B. die Division einer Ganzzahl und einer Zeichenkette, verhindert werden. Das zweite Prinzip wird als schwache Typisierung bezeichnet und erlaubt die implizite Konvertierung von Typen, wonach eine solche Division durchgeführt werden kann.

Ein Typsystem ist jener Bestandteil einer getypten Sprache, der in einem Programm die Typen aller Variablen und Ausdrücke überwacht. Nach der Zuordnung eines Ausdruckes zu einem Typ wird bei der statischen Überprüfung durch ein Typsystem festgestellt, inwiefern die Verwendung mit den enthaltenen Regeln verträglich ist [LR09]. Ein Programm wird nach CARDELLI [Car97] als *wohlgetypt* bezeichnet, wenn die Überprüfungen vollständig absolviert werden, andernfalls als *fehlgetypt*, falls die Abwesenheit von Fehlverhalten nicht garantiert werden kann. Für ungetypte Sprachen werden detaillierte Laufzeitbetrachtungen durchgeführt, um festgelegte Fehler aufzudecken.

Nach PIERCE [Pie02] sind Typsysteme in verschiedenen Anwendungsgebieten sinnvoll einsetzbar:

Fehlererkennung: Die wichtigste Eigenschaft der statischen Typüberprüfung ist das Aufspüren von Fehlern, die während der Programmierung erschaffen werden. Solche Fehler, die gewöhnlich durch die Analyse von Inkorrektheiten zur Laufzeit aufgedeckt werden, können früher aufgespürt und unmittelbar behoben werden.

Abstraktion/ Modularität: Eine weitere unterstützende Eigenschaft eines Typsystems ist das Durchsetzen einer disziplinierten Programmierung. Speziell in umfangreichen Systemen wird eine strukturierte Modularisierung sowie Festlegung von Schnittstellen erreicht und letztendlich die Abstraktion verbessert.

Dokumentation: Typen dienen der Verbesserung des Programmverständnisses. Die Typdeklarationen in einem Methodenkopf oder in Schnittstellen erzeugen eine Art von Dokumentation, die im Gegensatz zu Quellcodekommentaren ihre Aktualität nicht verlieren.

Sicherheit der Programmiersprache: Jede höhere Programmiersprache, die eine eigene Abstraktion von Systeminstruktionen beinhaltet und für diese, auch in höherer Gestalt, die Integrität garantiert, wird als sichere Sprache bezeichnet [Car97]. Ist eine solche Absicherung nicht vorhanden, gilt die Programmiersprache als unsicher.

Effizienz: Die Steigerung der Effizienz durch einen Compiler begründet sich auf den ermittelten Typinformationen. Ein auf hohe Leistungsfähigkeit ausgerichteter Compiler ist während seiner Optimierungs- und Generierungsphasen stark von diesen Informationen abhängig.

Weiterführende/ untypische Anwendungen: Die Anwendungsbereiche von Typsystemen gehen inzwischen über die Programmierung und den Entwurf von Sprachen hinaus. Eine statische Typisierung bildet in der Programmiersprache Java den Kern des Modells für Computer- und Netzwerksicherheit. Weiterhin werden die Prinzipien in Werkzeugen zur Programmanalyse verwendet, um beispielsweise Pseudonymnamen zu untersuchen [OJ97].

PIERCE [Pie02] nimmt eine Unterteilung der Programmiersprachen anhand ihrer Eigenschaften vor (vgl. Tabelle 2.3). Dabei wird nach dem Vorhandensein eines Typsystems und dem damit verbundenen Prüfzeitpunkt unterschieden sowie die Sicherheit der jeweiligen Sprache und die Art der Typisierung als Einteilungskriterien genutzt.

	statisch/ getypt	dynamisch/ ungetypt
sicher/ stark getypt	Haskell, Java, etc.	Lisp, Scheme, Perl
unsicher/ schwach getypt	C, C++, etc.	

Tabelle 2.3: Klassifizierung von Programmiersprachen [Pie02])

Die Programmiersprache C besitzt nach der Klassifizierung in Tabelle 2.3 die Eigenschaften: statisch/ getypt und unsicher/ schwach getypt:

- Aus der Einteilung in statisch/ getypt leitet sich für C ab, dass in der Sprache verschiedene Typen benutzt werden, die jeweils einen abgegrenzten Wertebereich besitzen. Die Ermittlung von Typfehler wird statisch durchgeführt, wodurch diese ausschließlich in der Phase der Übersetzung ermittelt werden.
- Für C gelten ebenso die Eigenschaften: unsicher und schwach getypt. Die Unsicherheit resultiert aus der fehlenden Integrität der vorhandenen oder erschaffbaren Abstraktionen bezüglich der darunterliegenden Speicherstrukturen. Spezielle Funktionalitäten, wie Zeigerarithmetiken und Typumwandlungen, sind unsicher, aber in der Benutzung stark verbreitet [Car97]. Jeder Speicherbereich kann als ein gewünschter Typ interpretiert werden [CR99]. Weiterhin wird C als schwach getypt bezeichnet, da das Auftreten eines Fehlers, welcher nicht den sofortigen Abbruch des Programms auslöst, möglich ist [Pie02]. Beliebige Speicherbereiche können als Daten eines Objekts behandelt werden, sodass beispielsweise auf Daten zugegriffen werden kann, die hinter dem Ende einer Struktur liegen [LR09].

PIERCE [Pie02] argumentiert, dass für die Programmiersprache C keine vollständige Sprachdefinition vorliegen kann, um für jedes Programm ein spezifisches Verhalten zu prognostizieren. Das Verhalten steht in direkter Abhängigkeit zur Speichernutzung des jeweiligen Compilers, wonach Unsicherheiten, wie nicht geprüfte Feldzugriffe oder Zeigerarithmetiken, abhängig von der Positionierung der Speicherbereiche sind. Demnach wird für C eine Prüfung des Quellcodes beziehungsweise der Typen durchgeführt. Wenn für ein Programm die Typsicherheit festgestellt wird, können die Eigenschaften der Programmiersprache C dennoch während der Laufzeit zu Fehlverhalten führen.

Innerhalb des C-Quellcodes sind trotzdem Untersuchungen durchführbar, die Typfehler feststellen und ein Fehlverhalten der Anwendung verhindern können. Der Fokus dieser Arbeit ist auf Referenzen gerichtet, wobei der Zugriff auf ein Quellcodeelement nur möglich ist, wenn es auch vorher erschaffen wurde. Eine Funktion ist nur nach vorheriger Definition/ Deklaration aufrufbar. Weitere Elemente, wie Variablen, müssen vor dem ersten Zugriff existieren. Eine Variable repräsentiert dabei einen elementaren Datentyp oder aber abgeleiteten Typ, der eine komplexe Zusammensetzung beinhaltet. Jeder Nutzung eines nicht elementaren Typs muss eine entsprechende Definition vorausgehen, die auch die verwendbaren inneren Elemente festlegt. Die Analyse von Referenzen stellt eine Beziehung zwischen dem Erschaffen und der Nutzung eines solchen Quellcodebestandteils her.

Im weiteren Verlauf der vorliegenden Arbeit wird ein System zur Analyse von Quellcodereferenzen im Kontext der Konditionen des CPP konzeptioniert und implementiert. Die Analyse ermöglicht die Realisierung eines produktlinienorientierten Typsystems, das die Referenzen eines Quellcodedokumentes für alle generierbaren Varianten auf Typfehler untersucht. Hierfür werden die Referenzen ermittelt, paarweise gegenübergestellt und hinsichtlich ihrer Abhängigkeit vom CPP verglichen. Aus den Untersuchungen können Aussagen über die Existenz von Typfehlern in bestehenden Anwendungen abgeleitet werden.

KAPITEL 3

Konzept für die CPP-spezifische Analyse

Das Ziel der vorliegenden Arbeit ist die Entwicklung eines produktlinienorientierten Typsystems, das die Referenzen eines C-Dokumentes im jeweiligen Präprozessorkontext auf Typfehler untersucht. Hierbei werden alle generierbaren Projektvarianten geprüft und Konfigurationsfehler aufgedeckt.

Zur Erfüllung des Vorhabens wird das Werkzeug *TypeChef* (Type Checking #iifdef) konzeptioniert und implementiert (vgl. Kapitel 4). Die einzelnen Phasen der Referenzprüfung sind in Abbildung 3.1 dargestellt. Jede der Teilphasen symbolisiert einen speziellen Abarbeitungsschritt des Werkzeuges. Ausgehend von einem C-Dokument werden in den aufeinander aufbauenden Schritten vorbereitende Veränderungen an der Dokumentstruktur vorgenommen, ein AST extrahiert, die Referenzanalyse durchgeführt und die referenzspezifischen Annotationen auf einen Typfehler untersucht.

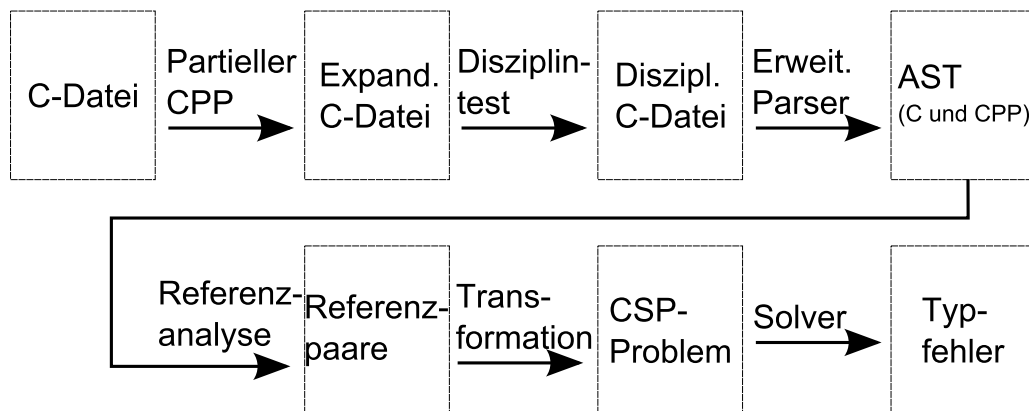


Abbildung 3.1: Übersicht der TypeChef-Phasen

In jeder Phase von *TypeChef* wird eine bestimmte Funktionalität realisiert. Die verschiedenen Eigenschaften des Werkzeuges und die damit verbundenen Teilaufgaben während der statischen Referenzanalyse gliedern sich wie folgt:

Partieller C-Präprozessor: Der Ausgangspunkt von *TypeChef* ist ein Quellcodedokument der Programmiersprache C, das je nach Umfang der variablen Anpassung an verschiedene funktionelle Eigenschaften mit den Direktiven des CPP durchgesetzt ist. Der partielle Präprozessor ist eine wichtige Phase zur Vorbereitung der Referenzanalyse. Innerhalb dieses Arbeitsschrittes wird der Quellcode gezielt vorverarbeitet und alle Direktiven hinsichtlich der damit formulierten Funktionalität

aufgelöst, ohne jedoch durch die bedingte Übersetzung angepasst zu werden. Diese spezielle Phase verändert das Dokument hinsichtlich komplexer Aufgaben, wie Makroersetzung und Dateiinklusion, ohne eine bestimmte Konfiguration festzulegen (vgl. Abschnitt 3.1).

Disziplintest: Der CPP erlaubt die variable Gestaltung des Quellcodes bis auf die Ebene von einzelnen Zeichen. Die Festlegung einer Disziplin für die verwendbaren Annotationen begrenzt die nutzbare Granularität. Diese Einschränkung legt das Auftreten einer CPP-Direktive klar fest und definiert das Spektrum der annotierbaren Quellcodeelemente (vgl. Abschnitt 3.2).

Erweiterter Parser: Der erweiterte Parser untersucht die expandierte und disziplinierte C-Datei und extrahiert einen abstrakten Syntaxbaum (engl. Abstract Syntax Tree (AST)). Der Parser für C-Quellcode wird um ein Regelwerk für die Anweisungen der bedingten Übersetzung erweitert. Der generierte AST enthält zusätzlich die Präprozessorabhängigkeit und bildet diese auf die enthaltenen Quellcodeelemente ab (vgl. Abschnitt 3.3).

Referenzanalyse: Die Phase der Referenzanalyse durchläuft den im vorherigen Schritt extrahierten AST. Anhand verschiedener Muster werden die Referenzen ermittelt und in einer erweiterten Symboltabelle festgehalten. Für jedes einzutragende Element werden außerdem die CPP-Hierarchie und somit die Bedingung für das Erreichen des Quellcodebereiches dokumentiert. Aus der Symboltabelle werden Referenzpaare zusammengestellt, die für ein Element die Einträge für das Erstellen sowie den Zugriff gegenüberstellen. Jedes dieser Paare wird für alle generierbaren Konfigurationen auf das Vorhandensein eines Typfehlers untersucht, der bei einem Zugriff auf ein Element ohne dessen vorherige Erstellung entsteht (vgl. Abschnitt 3.4).

CSP-Transformation und -Auswertung: Der Funktionsumfang des CPP erlaubt die Formulierung von komplexen Makrogleichungen, um die Variabilität zu definieren (vgl. Abschnitt 2.1.3). Aus einem zu untersuchenden Referenzpaar wird die CPP-Hierarchie der betrachteten Elemente in die Bedingungen eines CSP transformiert und hinsichtlich eines Typfehlers geprüft. Ein CSP-Solver ist ein Lösungssystem, das eine gegebene Bedingung auf Erfüllbarkeit testet. Aus einem Referenzpaar werden die Bedingungen für den Verbleib im Dokument in einer Aussage kombiniert, die einen Typfehler beschreibt. Falls eine Anwendung über ein Merkmalmodell verfügt, werden die darin formulierten Abhängigkeiten beachtet und in die zu prüfende Aussage aufgenommen. Der Solver sucht nach einer möglichen Variablenbelegung und gleichzeitig einer Konfiguration, für die ein fehlerhaft ausgeführter Zugriff auf ein Element resultiert. Sobald der Solver zu einer Lösung gelangt, ist das Paar nicht korrekt konfiguriert und ruft in einer gewissen Anzahl an Varianten ein Fehlverhalten hervor (vgl. Abschnitt 3.5).

Die einzelnen, in diesem Abschnitt skizzierten Phasen von *TypeChef* werden im weiteren Verlauf des Kapitels detailliert betrachtet und die Arbeitsweise der jeweiligen Teilkonzepte erläutert.

3.1 Partieller C-Präprozessor

Die Durchsetzung des C-Quellcodes mit allen Direktiven des C-Präprozessors stellt eine große Hürde für die statische Referenzanalyse dar. Die Komplexität der CPP-Anweisungen und die daran gekoppelte Transformation einer Datei müssen vollständig durch *TypeChef* beachtet werden, da neben der bedingten Übersetzung umfangreiche Umformungen vorgenommen werden können. Im Fokus der Analysen in *TypeChef* steht die konfigurationsabhängige Anpassung eines C-Dokumentes. Darin sind jedoch, neben den dafür verantwortlichen Anweisungen, weitere Direktiven vorzufinden. Hierbei werden unter anderem Dateien inkludiert oder Makros definiert. Die Menge der Direktiven, die nicht zur bedingten Übersetzung gehören, und vor allem deren Funktion während der Programmgenerierung sind nicht zu vernachlässigen. Eine vollständige Nachbildung eines C-Präprozessors ist auf Grund des enthaltenen Funktionsumfanges sehr aufwendig.

Ein wichtiger Beitrag der vorliegenden Arbeit ist die *partielle Ausführung eines C-Präprozessors*. Als Voraussetzung für die Analyse der Konfigurationsbedingungen wird eine Möglichkeit benötigt, um die dazugehörigen Textpassagen vor der Präprozessorverarbeitung zu schützen. Dieser Vorgang wird mittels eines eigenen Schrittes in *TypeChef* realisiert. Der partielle C-Präprozessor vollzieht eine angepasste Quellcodetransformation, die alle Eigenschaften des CPP nutzt und die Funktionalität der Direktiven umsetzt, jedoch die Konfiguration des Dokumentes verhindert. Folglich werden Makros ersetzt, gekapselte Dateiinhalte eingefügt und alle verbleibenden Direktiven verarbeitet. Das Resultat ist ein vollständig expandiertes Dokument in dem alle Abhängigkeiten hinsichtlich der CPP-Transformation aufgelöst sind und dennoch alle variabel zu gestaltende Quellcodebereiche im Kontext der Direktiven der bedingten Übersetzung erhalten bleiben.

In der Phase des partiellen Präprozessors werden durch *TypeChef* intern die anschließend aufgeführten Schritte bewerkstelligt:

Herauslösen der Konfigurationsbedingungen: In Vorbereitung auf die Ausführung des CPP werden alle Anweisungen der bedingten Übersetzung aus dem Quellcode auskommentiert. Demnach ist das Dokument nicht durch die konfigurationsbezogenen Transformationen betroffen.

Ausführung der Präprozessorverarbeitung: Das speziell angepasste Dokument wird vom CPP vollständig verarbeitet. Hierbei wird der Quellcode, wie ursprünglich vorgesehen, umstrukturiert. Jede genutzte CPP-Funktionalität wird folgerichtig im Dokument angewendet. Die in den Kommentaren hinterlegten Direktiven werden nicht ausgewertet und somit nicht verarbeitet.

Reintegration der Variabilitätsdirektiven: Abschließend muss der Konfigurationskontext wiederhergestellt werden, um im weiteren Ablauf von *TypeChef* untersucht zu werden. Die im ersten Schritt durchgeführten Änderungen werden aufgehoben und somit alle Makroabhängigkeiten rekonstruiert.

3.1.1 Beispiel

Die partielle Anwendung des C-Präprozessors wird im Folgenden an einem Quellcodebeispiel aus der Anwendung Femto OS¹ erläutert. Die Abbildung 3.2 zeigt den Ausgangszustand des Beispielabschnittes. In der ersten Zeile wird unter dem Namen *Tchar* ein Makro für den Ersatztext *unsigned char* angelegt. Die Definition der Funktion *genLogOs* ist mit einer *#if*-Direktive annotiert, wodurch der eingeschlossene Bereich nur dann präsent ist, wenn die formulierte Bedingung *includeGenLogOs == cfgTrue* eintritt. In der Signatur der Funktion ist das im Vorfeld definierte Makro enthalten und wird als ein Parameter verwendet.

```

1 #define Tchar unsigned char
2 ...
3 #if (includeGenLogOs == cfgTrue)
4     void genLogOs(void (*pipe)(Tchar)){...}
5 #endif
6 ...

```

Abbildung 3.2: Ursprungszustand des Quellcodes

Das Ergebnis der Abarbeitung durch den partiellen C-Präprozessor ist in Abbildung 3.3 dargestellt. Der enthaltene Quellcodeabschnitt zeigt C-Quellcode, der hinsichtlich aller nicht zur Konfiguration verwendeten Direktiven transformiert ist. In dem verwendeten Beispiel wird das Makro *Tchar* durch den Ersatztext *unsigned char* substituiert. Weiterhin wird gezeigt, dass die Zeilen 2 und 4 nicht verarbeitet wurden und die Funktion *genLogOs* (Z. 3) somit in Abhängigkeit von der umschließenden Direktive konfiguriert wird.

```

1 ...
2 #if (includeGenLogOs == cfgTrue)
3     void genLogOs(void (*pipe)(unsigned char)){...}
4 #endif
5 ...

```

Abbildung 3.3: Ergebnis des partiellen Präprozessors

3.1.2 Include Guards

Das Herauslösen der bedingten Übersetzung muss hinsichtlich eines Direktivenkonstruktes eingeschränkt werden. Die Include Guards werden in den Beschreibungsdateien als Schutz vor mehrfachem Einfügen des Inhaltes verwendet. Die Abbildung 3.4 (S. 29) zeigt ein Beispiel für diesen Schutzmechanismus.

Die fehlende Definition eines speziellen Makros wird als Indikator für die Inklusion genutzt (Z. 1). Wie in Zeile 2 dargestellt, wird das Makro im direkten Anschluss definiert, wodurch der Inhalt des Header-Files bei der Verarbeitung jeder weiteren *#include*-Direktive nicht eingefügt wird. Das Herauslösen unterscheidet zwischen der

¹ <http://www.femtoos.org/>

```
1 #ifndef FEMTOOS_CORE_H
2 #define FEMTOOS_CORE_H
3 ...
4 #endif
```

Abbildung 3.4: Einfügeschutz eines Header-Files (Femto OS)

Nutzung von Include Guards und den sonstigen Konfigurationsdirektiven, um der Einfügeschutz nicht aus der CPP-Verarbeitung auszuschließen. Somit kann die korrekte Funktionsweise des Mechanismus gewährleistet werden.

Die partielle Nutzung des C-Präprozessors ist eine essenzielle Phase von *TypeChef*. Für die Analyse von C-Quellcode im Zusammenhang mit der enthaltenen Variabilität wird ein schwieriges Problem gelöst. Die Arbeitsweise des partiellen CPP beugt der Notwendigkeit einer vollständigen Bearbeitung aller verbleibenden Direktiven im Quellcode vor. Folglich wird ein Nachbilden des CPP-Funktionsumfanges vermieden und durch ein dafür ausgelegtes Werkzeug ausgeführt. Letztendlich hat die Verwendung des partiellen C-Präprozessors für die weiterführenden Analysen eines C-Projektes zur Folge, dass alle benötigten Dateien inkludiert und in diesem Zusammenhang die richtige Reihenfolge und Hierarchie beachtet werden. Des Weiteren werden alle Makros gemäß ihrer Definition beziehungsweise des hinterlegten Ersatztextes substituiert. Unter Verwendung des CPP wird die notwendige Abarbeitungsreihenfolge auch für die Textersetzung eingehalten und eine geschachtelte Makroverwendung folgerichtig aufgelöst.

3.2 Disziplintest

Das Anwendungsspektrum für die Annotation von Quellcodebestandteilen mit Präprozessordirektiven ist vielfältig und kann auf unterschiedlichen Ebenen der Granularität angewendet werden. Die Abarbeitung des CPP geschieht Token-basiert, sodass z. B. eine öffnende Klammer im Quellcode annotiert werden kann, wenn ein solches Vorgehen für die Konfiguration der Anwendung benötigt wird. Demnach skaliert die Granularität von einzelnen Zeichen und einfachen Anweisungen, wie Variablenzugriffe, über umfangreiche Konstrukte, wie komplette Methoden, bis zu vollständigen Dateien. Die Skalierbarkeit des CPP führt zu Annotationen auf verschiedenen Ebenen der Granularität und erlaubt eine flexible Anwendbarkeit für die bedingte Übersetzung von Codefragmenten.

Die mögliche Granularität stellt aber auch eine Hürde für die Untersuchung der Referenzpaare dar. Dementsprechend wird der Grad der in *TypeChef* analysierbaren Variabilität insofern eingeschränkt, dass das Auftreten von CPP-Direktiven nur innerhalb eines festgelegten Spektrums an Granularität zulässig ist. Die Einschränkung verringert den Umfang der möglichen Anwendbarkeit des CPP. Jedoch werden gleichzeitig Syntaxfehler vermieden, die durch das Entfernen des umschlossenen Quellcodebereiches entstehen. Die Begrenzung bewirkt eine Erleichterung der Analyse, da die Quellcodeelemente beziehungsweise die spätere Repräsentation durch einen Teilbaum

im AST komplett einer Variabilitätsbedingung zugeordnet werden können und nicht zusätzlich deren Integrität bezüglich der fehlerfreien Zusammensetzungen, speziell die syntaktische Korrektheit, sicherzustellen ist [KAK08, KAK09]. Daraus resultiert die Festlegung einer Disziplin für die Verwendung von Annotationen.

Mit Hilfe dieser Einschränkungen wird die kombinierte Analyse von C und CPP stark erleichtert. Die variable Anpassung von C-Quellcode führt zur Mischung der jeweils nutzbaren Sprachkonstrukte. Das Zusammenspiel ruft Mehrdeutigkeiten und Fehlzuordnungen im Analysesystem hervor, die mittels solcher Festlegungen verhindert werden können [Pad09].

Die Einteilung der Annotationen erfolgt hinsichtlich der Analysierbarkeit seitens *TypeChef* in die Klassen: diszipliniert und undiszipliniert [Käs10]. Im weiteren Verlauf des Abschnittes werden diese Klassen näher betrachtet und mit Hilfe von Quellcodebeispielen erläutert, wie für *TypeChef* die Unterscheidung zwischen diesen beiden Klassen vorgenommen wird.

3.2.1 Disziplinierte Annotationen

Eine Markierung wird als *diszipliniert* bezeichnet, wenn sie auf einer bestimmten Stufe der Granularität wirkt und ein Element bis zu der Ebene von Anweisungen vollständig umschließt. Aus Begrenzung der möglichen Granularität resultiert die Festlegung einer unteren Schranke für die kombinierte Analyse von C und CPP. Die in *TypeChef* verwendete Grammatik wird zur Unterstützung von disziplinierten Annotationen erweitert. Eine Unterstützung von weiteren Stufen der Präprozessorgranularität bedarf einer umfangreichen Durchsetzung der C-Grammatik hinsichtlich der CPP-Erweiterung. Jedoch kann für eine solche Veränderung keine Korrektheit der ursprünglichen Funktionalität sichergestellt werden.

Für das im Rahmen dieser Arbeit entstehende Werkzeug wird die Disziplin dahingehend festgelegt, dass ein oder mehrere Elemente auf der obersten Quellcodeebene durch Direktiven umschlossen werden können. Hierzu gehören die Definition beziehungsweise die Deklaration von global anwendbaren Elementen, wie Funktionen, Strukturen/Unions, Typdefinitionen oder Variablen. Weiterhin werden Annotationen von einzelnen oder mehreren Statements innerhalb einer Funktion oder von Komponenten bei der Definition einer Struktur/ Union als diszipliniert bezeichnet. Das Auftreten einer variablen Anpassung auf einer Ebene feinerer Granularität oder ringsum von Bestandteilen eines Elementes wird durch *TypeChef* als undiszipliniert angesehen.

Ein umschlossener Bereich kann die zu betrachtenden Elemente auch mehrfach beinhalten, woraus für jedes die Abhängigkeit von der annotierenden Bedingung folgt. Die markierbaren Fragmente können auf vielfältigen Ebenen des Dokumentes vorkommen, wo sie erschaffen werden beziehungsweise ein Zugriff auf sie realisiert wird. Die Ebenen werden in die Klassen *global* und *lokal* eingeteilt und im Folgenden näher beleuchtet. Unter der Verwendung von Quellcodebeispielen wird aufgezeigt, welche globalen und lokalen Elementannotationen unterstützt werden.

Globale Ebene

In einer Datei der Sprache C können Quellcodeelemente auf der globalen Ebene angelegt werden. Auf der oberen Ebene werden Funktionen, globale Variablen und synonyme Typen sowie Strukturen/ Union und Aufzählungen erzeugt. Die Einordnung solcher Elemente in eine Konfiguration erfolgt durch das Umschließen der betroffenen Zeilen oder Blöcke mit einer bedingungsdefinierenden Präprozessoranweisung und einer notwendigen `#endif`-Direktive.

Zur Illustration des Auftretens dieser Art von Elementannotationen dienen die in Abbildung 3.5 dargestellten Ausschnitte aus Femto OS. In den Zeilen 2 und 4 wird jeweils eine globale Variable erzeugt. Die Anweisungen werden durch Direktiven des CPP markiert und sind folglich nur in bestimmten Konfigurationen präsent und nutzbar. Im Speziellen ist die Variable *Tuint08 uiLastOsStatus* (Z. 4) von der Verschachtelung der CPP-Anweisungen (Z. 1, 3) abhängig. Weiterhin ist im Beispiel die Funktionsdefinition *privShowError* dargestellt (Z. 9-10). Der dazugehörige Quellcodebereich ist vollständig umschlossen und nur unter der Voraussetzung einer erfüllten `#if`-Direktive enthalten (Z. 8-11). Abschließend zeigt die Abbildung in den Zeilen 13 bis 20 das Anlegen einer Struktur, die in Folge der Bedingung von einem der Makros *cfgCheckTaskStack* und *cfgCheckRegisters* abhängig ist.

```
1 #if (cfgIntUserDefined == cfgTrue)
2     Tuint16  uiIsrLoadTotal;
3     #if (cfgIntOsProtected == cfgTrue) && (cfgUseLowPowerSleep == cfgFalse)
4         Tuint08  uiLastOsStatus;
5     #endif
6 #endif
7
8 #if (defCheckReportingError == cfgTrue)
9 static void privShowError(Tuint08 uiMessage, Tuint08 uiCallId, Tuint08 uiInfo)
10 { ... }
11 #endif
12
13 #if (cfgCheckTaskStack == cfgTrue) || (cfgCheckRegisters == cfgTrue)
14     typedef struct {
15         Tuint08  r31;
16         Tuint08  r30;
17         ...
18         Tuint08  uiRegisterUse;
19         ...
20     } TtaskSave;
21 #endif
```

Abbildung 3.5: Annotation von globalen Elementen (Femto OS)

Lokale Ebene

Der betrachtete Einsatzbereich von Präprozessordirektiven erstreckt sich außerdem auf die lokale Quellcodeebene. Dementsprechend findet eine Verlagerung der konfigurationsabhängigen Anpassungen in den gekapselten Bereich der Elemente: Funktion, Struktur und Union statt. Für eine Funktion bedeutet diese Möglichkeit eine variable Zusammenstellung der involvierten Anweisungen, wodurch insgesamt die darin realisierte Funktionalität angepasst werden kann. Ein solches Prinzip ist ebenso für Struk-

turen und Union anwendbar. Der innere Aufbau, speziell die Komponenten, werden mit CPP-Direktiven umrahmt, sodass die jeweilige Eigenschaft nicht in jeder Variante verfügbar ist.

Die Quellcodebeispiele in Abbildung 3.6 entstammen erneut aus der Anwendung Femto OS. Im oberen Teil der Darstellung ist die Definition der Funktion *privShowError* enthalten. Innerhalb des dazugehörigen Blockes existieren verschiedene annotierte und nicht annotierte Elemente. Der innere Bereich wird in die Abschnitte Deklarations- und Anweisungsliste unterteilt. In den beiden Listen können die jeweiligen Bestandteile mit Direktiven in verschiedenartigen Kombinationen versehen werden. Das Anlegen der Variable *cfgCheckAlwaysFatal* (Z. 3) sowie der Funktionsaufruf in Zeile 9 gehören nicht zu einem markierten Quellcodeabschnitt und sind somit in jeder Variante präsent. Speziell werden hier die Variablen in den Zeilen 5-6 in Abhängigkeit von dem Makro *cfgCheckAlwaysFatal* erschaffen und sind dann ausschließlich in diesem Kontext nutzbar. Weiterhin ist der Aufruf der Funktion *privEnableGlobalInterrupts* (Z. 11) als konfigurierbar markiert und wird bei der Erfüllung der umschließenden Bedingung (Z. 10) integriert.

```

1 static void privShowError(Tuint08 uiMessage, Tuint08 uiCallId, Tuint08 uiInfo)
2 {
3     Tuint08 cfgCheckAlwaysFatal = (uiMessage & errMessageGetMask);
4     #if (cfgCheckAlwaysFatal == cfgTrue)
5         const Tbool bFatal = true;
6         const Tbool bReturn = false;
7     #else
8         ...
9     #endif
10    ...
11    portSetupTimerInterrupt();
12    #if (cfgInt0sProtected == cfgFalse)
13        privEnableGlobalInterrupts();
14    #endif
15 }
16
17 typedef struct
18 {
19     Tstack    pcStackLevel;
20     Tuint08   uiTaskStatus;
21     #if (defUseDelay == cfgTrue)
22         Ttick    uxDelay;
23     #endif
24     ...
25 } TtaskExtendedControlBlock;

```

Abbildung 3.6: Annotation von lokalen Elementen (Femto OS)

Im unteren Beispiel wird die Struktur *TtaskExtendedControlBlock* definiert (Z. 15-23). Die darin hinterlegten Variablen besitzen verschiedene Datentypen und werden in der Funktion von Strukturkomponenten zu einer eigenständigen Einheit zusammengefasst (Z. 17-21). Auf dieser Ebene der Variabilität sind Annotationen zulässig, die eine Komponente, wie in den Zeilen 19-21 gezeigt, variabel gestalten. In jeder Variante, in der die Bedingung *defUseDelay == cfgTrue* erfüllt ist, besitzt die Struktur mit *Ttick uxDelay* eine zusätzlich nutzbare Eigenschaft. Die Präprozessormarkierungen können auch mehrere Komponenten umschließen und eine beliebige Verschachtelung von Direktiven erzeugen.

3.2.2 Undisziplinierte Annotationen

Alle feingranularen Annotationen, die in ihrer Anwendung die Zusammensetzung eines Elementes verändern, werden als *undiszipliniert* angesehen. Zu dieser Menge von markierten Quellcodefragmenten gehören z. B. einzelne Zeichen, wie eine schließende Klammer oder konfigurationsabhängige Rückgabetypen in Funktionsdefinitionen.

Undisziplinierte Annotationen stellen potenzielle Fehlerquellen dar, da der umschlossene Quellcode im Zuge der Texttransformation des Präprozessors entfernt wird und beispielsweise ein Syntaxfehler, wie eine fehlende schließende Klammer, verursacht wird.

Solche Fehler hervorrufenden Konstrukte können in verschiedenen Quellcodeelementen enthalten sein. Die Abbildung 3.7 zeigt Quellcodebeispiele für Annotationen, die in einer Programmvariante zu Fehlern führen. Im linken Teil wird eine Funktion mit annotiertem Rückgabewert gezeigt. Hierbei entsteht eine inkorrekte Struktur, wenn die Direktiven (Z. 1-6) gleichzeitig wahr oder falsch werden und somit zwei beziehungsweise keine Rückgabewerte festgelegt werden. Der rechte Teil zeigt eine variable Alternativverzweigung einer if-Anweisung, wobei die schließende Klammer des vorangehenden Blockes auch umschlossen wird (Z. 4-7). Ein Fehler tritt auf, wenn dieser Bereich entfernt wird, da dann auch die Klammer fehlt.

<pre> 1 #if defined(X) 2 int 3 #endif 4 #if !defined(Y) 5 float 6 #endif 7 foo() 8 {...} </pre>	<pre> 1 if(...) 2 { 3 ... 4 #if defined(X) 5 } else 6 {...} 7 #endif </pre>
---	--

Abbildung 3.7: Undisziplinierte Annotationen

Die Verwendung des CPP zur Konfiguration einer Quelldatei wird von *TypeChef* nicht bis zur kleinstmöglichen Granularitätsstufe unterstützt. Eine Eingrenzung erfolgt auf diszipliniert angewendete CPP-Anweisungen, da die Unterstützung des vollen Präprozessorspektrums nicht möglich ist. Die dafür notwendigen Anpassungen verändern die bestehende Grammatik in sehr hohem Maße. Folglich kann eine korrekte und vollständige Funktionsweise des Analysesystems nicht sichergestellt werden.

Eine solche Einschränkung benötigt für undisziplinierte Annotationen eine Möglichkeit zur Herstellung der definierten Disziplin. Für die mit *TypeChef* durchzuführenden Analysen wird die Disziplin manuell hergestellt.

3.2.3 Manuelle Disziplinierung von Annotationen

Die Einschränkung der Untersuchungen auf disziplinierte Markierungen ist prinzipiell möglich, da der Anteil an Fehler verursachenden Direktivenblöcken gering ausfällt und mittels manueller Umformungen in eine verständliche Form überführt werden kann [BM01, Käs10].

BAXTER [BM01] argumentiert, dass ungefähr 15 Prozent des betrachteten Quellcodes nicht von dem verwendeten Parser erkannt werden und bezeichnet die möglichen Fälle als unstrukturierte Benutzung des Präprozessors. Weiterhin wird die notwendige Restrukturierung der nicht verständlichen Fragmentmenge als schnell durchzuführende Aufgabe angesehen.

In KÄSTNER [Käs10] werden bestehende C-Anwendungen in Bezug auf die vorzufindende CPP-Disziplin bei Elementen auf der obersten Quellcodeebene untersucht. Etwa 86 Prozent aller Markierungen werden als diszipliniert eingestuft. Auch hier wird der Aufwand für die Überführung der restlichen Annotationen als vertretbar bezeichnet.

Die automatische Disziplinierung eines Quellcodedokumentes steht nicht im Fokus dieser Arbeit. Die Realisierung eines solchen Mechanismus wäre jedoch möglich. Hierbei müssen die Konfigurationsanweisungen der betroffenen Bereiche auf die nächstmögliche, unterstützte Quellcodeebene verschoben werden. An dieser Stelle ist zu beachten, dass der ursprüngliche Kontext erhalten bleibt beziehungsweise durch die Erweiterung der Annotationen oder der Quellcodestruktur wiederhergestellt wird.

3.3 Erweiterter Parser

Im Fokus der vorliegenden Arbeit steht der Einsatz der Direktiven des CPP in bestehenden C-Projekten. In diesem Zusammenhang werden speziell die Anweisungen der bedingten Übersetzung und die daraus resultierende Erschaffung von Variabilität untersucht. Die Kombination der beiden Sprachen beziehungsweise der Sprachkonstrukte führt zu einer erschwerten Analyse und bedarf eines Systems, welches Regeln für die Verschränkungen definiert und demnach Untersuchungen ermöglicht. Nach PADIOLEAU [Pad09] sind C und CPP getrennt voneinander einfach zu parsen, jedoch müssen tiefgreifende Veränderungen vorgenommen werden, um die CPP-Direktiven in eine Grammatik für C zu integrieren. Des Weiteren entstehen neu hervorgerufene Probleme, wie z. B. Mehrdeutigkeiten, die bei der kombinierten Analyse von C und CPP hervorgerufen werden.

In dieser Arbeit wird das Vorhaben verfolgt, ein Analysewerkzeug für C-Anwendungen, in denen CPP-Direktiven eine konfigurationsbezogene Anpassung hervorrufen, zu erschaffen. Grundlegend wird eine Komponente benötigt, die aus einem Quellcodedokument alle benötigten Informationen extrahiert. Innerhalb des Vorganges werden die enthaltenen Sprachelemente und die strukturelle Zusammensetzung der Datei in Abhängigkeit von den verwendeten Konfigurationsbedingungen herausgestellt. Zur Weiterverarbeitung wird als Analyseergebnis eine Struktur benötigt, die sich zur Darstellung der Informationen eignet und die Struktur des Quellcodes abbildet. Durch einen AST kann eine solche Abbildung erreicht werden, wodurch die Dokumentstruktur in Bezug auf vorhandene Annotationen repräsentiert wird.

Der beschriebene Vorgang wird in Abbildung 3.8 (S. 35) zusammengefasst. Dabei wird für die C-Präprozessor-basierte Quellcodeannotation aufgezeigt, welche Informationen zusätzlich in den AST integriert werden. Von besonderem Interesse sind hierbei die Konditionen, mit denen verschiedene Quellcodebestandteile bezüglich einer spezifischen Konfiguration versehen werden sowie die Integration dieser Abhängigkeiten in die resultierende Baumstruktur.

```

1  #if !defined(ALPHA)
2      #if defined(pyr)
3          char * buffer;
4      #endif
5  #elif defined(i386)
6      int count;
7  #endif

```

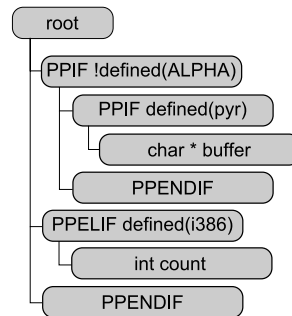


Abbildung 3.8: Erweiterung des AST um CPP-Direktiven

Wie in Abbildung 3.8 dargestellt, wird für jede der ersten beiden Direktiven ein Knoten *PPIF* erzeugt, der die vorherrschende Bedingung modelliert. Der Knoten *PPENDIF* beschreibt den Abschluss der inneren Direktive, sodass bis zum nächsten Auftreten eines *#endif* oder einer *#elif*/*#else*-Verzweigung, welche im AST als *PPELIF*/*PPELSE*-Knoten dargestellt werden, alle Elemente der Bedingung der ersten Direktive unterliegen. Ausgehend von dieser Art der Informationsaufbereitung dient der AST als Quelle für weiterführende Untersuchungen.

3.4 Referenzanalyse

Das Analysesystem von *TypeChef* stellt eine Verbindung zwischen den Quellcodeelementen der Programmiersprache C und den Direktiven der bedingten Übersetzung her. Weiterhin werden die existierenden Referenzbeziehungen ermittelt und die Paare zwischen den elementspezifischen Operationen gebildet.

3.4.1 Referenzen ermitteln

Der AST ist die Hauptquelle für *TypeChef*, um die Informationen für alle zu untersuchenden Quellcodeelemente zu ermitteln. Die Programmiersprache C setzt sich aus verschiedenen Bestandteilen zusammen. Mit Hilfe der Elemente Variable, Funktion, Aufzählung, Struktur/ Union und Typdefinition wird einem Speicherbereich eine elementare, vordefinierte beziehungsweise auch abgeleitete Interpretation zugewiesen. Die Festlegung geschieht durch einen eindeutigen *Bezeichner*, der für das Element, in Kombination mit der hinterlegten Interpretation, die weitere Verarbeitung ermöglicht. Die Bestandteile besitzen jeweils spezifische Operationen, die ein Element des Typs

erschaffen oder einen Zugriff realisieren. Die hierbei nutzbaren Operationen werden anhand von speziellen Mustern im AST erkannt und in eine Symboltabelle eingetragen.

Mit Hilfe von *TypeChef* werden Referenzen im Kontext des CPP auf die Korrektheit aller Konfigurationen untersucht. Der intern verwendete AST enthält neben den verschiedenen Elementen auch die Direktiven der bedingten Übersetzung. Folglich wird die Abhängigkeit eines Untersuchungsobjektes durch die umschließende Hierarchie aus CPP-Anweisungen beschrieben.

Auf den Konfigurationsabhängigkeiten beruhen alle weiteren in *TypeChef* durchgeführten Auswertungen. Somit wird die Symboltabelle erweitert und zu jedem Eintrag die Präprozessorbedingung, die den Verbleib im Quellcode formuliert, hinzugefügt. Die Abbildung 3.8 zeigt einen AST, der die annotierte Deklaration der Variablen *buffer* und *count* erfasst. Die intern verwaltete Symboltabelle wird anhand aller notwendigen Informationen aufgebaut (vgl. Tabelle 3.1). Für die beiden Variablen wird der Bezeichner und die Operation, hier eine Variablendeklaration sowie der Gültigkeitsbereich und die Präprozessorbedingung ermittelt und ein Eintrag erzeugt. Die Operation ist dabei das spezifische Erstellen eines Elementes. In C existieren weiterhin die Definition/ Deklaration von Funktionen und die Definition der Elemente Struktur/ Union, Aufzählungstyp sowie das Anlegen einer synonymen Typdefinition.

Name	Operation	Scope	CPP-Kondition
buffer	Vardekl.	0	!defined(ALPHA) \wedge defined(pyr)
count	Vardekl.	0	defined(ALPHA) \wedge defined(i386)

Tabelle 3.1: Erweiterte Symboltabelle

Die Einträge der Symboltabelle werden durch *TypeChef* zur weiteren Verarbeitung, speziell der Bildung von zu untersuchenden Paaren, eingesetzt. Jeder Eintrag repräsentiert die Operation zum Erschaffen des jeweiligen Elementes.

3.4.2 Referenzpaare ermitteln

Zwischen den verschiedenen Quellcodeelementen bestehen Beziehungen, die hinsichtlich des Kontextes der Annotation durch CPP-Direktiven potenzielle Quellen für Typfehler darstellen. In diesem Arbeitsschritt werden die aufbereiteten Informationen, die während der Traversierung des erstellten AST nach verschiedenen Vorgaben zusammengestellt werden, als Grundlage verwendet. Die erweiterte Symboltabelle umfasst alle Informationsobjekte und wird für die weiterführende Analyse von Referenzen genutzt. Nachfolgend werden die Zusammenhänge zwischen den verschiedenen Elementen hergestellt und in einem Referenzpaar formuliert:

Funktionen: Der Zugriff auf eine Funktion erfolgt unter Verwendung des Funktionsaufrufes. Für jeden gefundenen Aufruf ist die passende Definition/ Deklaration zu ermitteln und für das erschaffene Referenzpaar auf einen Typfehler zu untersuchen.

Durch den partiell ausgeführten Präprozessorschritt werden die notwendigen Einfügeoperationen durchgeführt, wobei eine inkludierte Header-Datei ausschließlich die Deklaration einer Funktion enthalten kann. Eine zu prüfende Referenzbeziehung besteht zwischen dem Aufruf und der Deklaration.

Variablen: Die Nutzung einer Variablen setzt deren Deklaration voraus und legt ein weiteres zu analysierendes Referenzpaar fest. Neben der Direktivenabhängigkeit ist der Gültigkeitsbereich zu beachten, um eine korrekte und eindeutige Zuordnung zu erreichen.

Ein Spezialfall ist der Funktionszeiger, da dieser hinsichtlich des Zugriffes einer Variablen gleicht, jedoch eine Verbindung zur Definition/ Deklaration erzeugt werden muss.

Struktur-/ Uniontyp: Ein Zugriff auf einen Struktur-/ Uniontyp wird beispielsweise bei der Deklaration von Variablen, der Ausführung eines Cast oder bei Typdefinitionen ermittelt. Hierbei wird eine Beziehung zur Definition des Elementes hergestellt und eine weitere Klasse von Referenzpaaren zusammengestellt.

Die Bestandteile von Struktur oder Union können durch den CPP annotiert werden und somit die Zusammensetzung hinsichtlich einer Konfiguration verändern. Im Rahmen der Referenzanalyse wird jeder vorhandene Komponentenzugriff auf mögliche Fehler im Konfigurationskontext untersucht.

Aufzählungstyp: Der Aufzählungstyp ist eine Ansammlung von Ganzzahlkonstanten, die eine instantiierte Variable dieses Typs annehmen kann. Für jeden Zugriff auf eine solche Konstante ist sicherzustellen, ob die passende Konstante in einem fehlerfreien CPP-Kontext existiert. Die verschiedenartigen Nutzungsmöglichkeiten eines Struktur-/ Uniontyps gelten ebenso für den Aufzählungstyp und beschreiben jeweils ein Paar mit der Definition dieses erweiterten Typs.

Typdefinition: Die Typdefinition erlaubt die Festlegung eines synonym nutzbaren Bezeichners. In den Referenzuntersuchungen werden die Verbindung zwischen der Definition des Ersatznamens und dessen Einsatz bei Deklarationen oder innerhalb von Anweisungen fokussiert und auf eine potenziell fehlerhafte Konfiguration getestet, falls eine Abhängigkeit zum CPP besteht.

3.4.3 Notwendigkeit der Analyse

Zwischen den Elementen eines Paares ist eine Beziehung herzustellen. Die Entscheidung über die Notwendigkeit einer näheren Untersuchung wird auf der Grundlage der Elementannotation getroffen.

Keine Annotationen/ Zugriff annotiert: Die Überprüfung auf einen Typfehler ist nicht erforderlich, wenn die beiden Elemente des Paares immer im Dokument enthalten sind und keine Annotation zugeordnet wird. Weiterhin ist ein mit Direktiven umschlossener Zugriff auf ein nicht markiertes Erschaffen eines Elementes in jeder konstruierbaren Quellcodevariante als fehlerfrei anzusehen, da der referenzierte Bestandteil immer vorhanden ist.

Erstellen annotiert/ Zugriff nicht annotiert: Die Konstellation aus dem annotierten Erstellen und dem in jeder Konfiguration ausgeführten Zugriff erzeugt, ohne Zusatzinformationen über das Projekt, grundsätzlich einen Typfehler, da der Quellcodebereich zur Elementerzeugung in einer geringeren Anzahl an Varianten nutzbar ist. Sobald eine zu untersuchende Anwendung ein Merkmalmodell besitzt, ist dieses in die Analyse einzubeziehen. Die darin beschriebenen Abhängigkeiten können für die betrachtete Konstellation festlegen, dass die erstellende Operation verbindlich. Demnach existiert der entsprechende Quellcodebereich in jeder Variante.

Beide Elemente annotiert: Sobald für beide Bestandteile des Paares eine Markierung durch den CPP festgestellt wird, muss die Elementkombination in einer auf den Informationen aufbauenden Direktivenanalyse hinsichtlich der Anfälligkeit für einen Typfehler untersucht werden (vgl. Kapitel 3.5).

Der nächste Abschnitt beschreibt, anhand von Beispielen aus der Anwendung Femto OS, die Notwendigkeit einer Untersuchung. Hierbei werden verschiedene Situationen aufgezeigt, die mit Hilfe von *TypeChef* analysiert werden können.

3.4.4 Beispiele

Für jedes Auftreten eines Referenzpaares stellt *TypeChef* die Zulässigkeit der Ausführung heraus. Die ermittelte Direktivenhierarchie legt für das Paar fest, unter welchen Gegebenheiten das Element im Quellcode vorhanden ist. Der Vergleich der beiden Gesamtbedingungen zeigt, ob eine Variante konstruiert werden kann, in der ein Typfehler hervorgerufen wird. Im weiteren Verlauf des Abschnittes werden Beispiele für Annotationen aus Femto OS sowie die dabei existierenden Beziehungen aufgeführt.

In der Abbildung 3.9 (S. 39) sind verschiedene Ausprägungen von annotierten Referenzpaaren im Kontext der CPP-Direktiven dargestellt. Die in Zeile 2 angelegte Variable *Tuint08 uiFsStatus* wird in der Steuerstruktur in Zeile 6 zur Entscheidungsfindung herangezogen. Bei dem Vergleich der Annotationen wird ersichtlich, dass die Elemente durch die gleiche Voraussetzung konfiguriert werden und somit kein Typfehler existiert.

Im unteren Bereich der Darstellung wird die Funktion *privUnblockTask* unter der in den Zeilen 10 bis 12 vorzufindenden Bedingungen definiert. Der passende Funktionsaufruf (Z. 20) wird als **#else**-Fall ausgeführt, wenn die Bedingung der Direktive in Zeile 17 nicht zum Wahrheitswert 1 aufgelöst wird. In diesem Fall ist zu prüfen, ob die Bedingungen sich nicht widersprechen und der Aufruf nur dann im Quellcode verbleibt, wenn auch die Definition vorhanden ist. Existiert eine Variante, in der der Aufruf ohne die dazugehörige Definition im Code vorzufinden ist, besteht für die Verbindung der beiden Elemente ein Typfehler, woraus eine entsprechenden Fehler bei der Übersetzung resultiert.

Das in der Darstellung 3.10 (S. 39) gezeigte Beispiel ist die variable Zusammensetzung einer Struktur. In den Zeilen 5 bis 7 wird der Struktur die konfigurationsabhängige Komponente *Tuint08 uiTaskMonitor* hinzugefügt. Die Eigenschaft ist in allen Anwendungsvarianten vorhanden, in denen die **#if**-Direktive erfüllt wird. Im

```

1  #if (cfgUseFileSystem == cfgTrue)
2      Tuint08 uiFsStatus;
3  #endif
4  ...
5  #if (cfgUseFileSystem == cfgTrue)
6      if ((uiFsStatus & defFsBurnBlockGetMask) == defFsBurnBlockActive)
7          {...}
8  #endif
9  ...
10 #if (cfgUseSynchronization != cfgSyncNon) ||
11     (cfgUseFileSystem == cfgTrue) ||
12     (cfgUseEvents == cfgTrue)
13     static void privUnblockTask(Tuint08 uiControlTaskNumber)
14     {...}
15 #endif
16 ...
17 #if (cfgUsePrioritizedRelease == cfgTrue)
18 ...
19 #else
20     privUnblockTask(uiLoopTask | defParaLockStateKeep | defParaRetStateTrue);
21 #endif

```

Abbildung 3.9: Fehlerhafte Referenzen durch Variabilität 1 (Femto OS)

Bereich der Zeilen 11 bis 15 wird eine Variable mit dem Strukturtyp *TtaskControlBlock* initialisiert und zur Verzweigung (Z. 13) genutzt. Hierbei wird auf die annotierte Komponente, dessen Verbleiben im Quelltext von dem Makro *defUseTaskMonitor* (Z. 5) abhängig ist, verwendet. Der Zugriff geschieht unter der Bedingung, dass das Makro *cfgUseFileSystem* (Z. 12) in der Konfiguration enthalten ist. Folglich unterscheiden sich die jeweiligen Voraussetzungen für die Komponente und den Zugriff auf diese Eigenschaft. Für eine Konstellation, in der die Komponente nicht verfügbar ist, kann kein Aufruf durchgeführt werden und verursacht für dieses Beispiel einen Typfehler.

```

1  typedef struct{
2      Tstack pcStackLevel;
3      Tuint08 uiTaskStatus;
4      ...
5      #if (defUseTaskMonitor == cfgTrue)
6          Tuint08 uiTaskMonitor;
7      #endif
8      ...
9  } TtaskControlBlock;
10 ...
11 TtaskControlBlock * taskTCB = privTcbList(uiTaskNumber);
12 #if (cfgUseFileSystem == cfgTrue)
13     if ( (taskTCB->uiTaskMonitor ...))
14         {...}
15 #endif

```

Abbildung 3.10: Fehlerhafte Referenzen durch Variabilität 2 (Femto OS)

Die Auswertung der mit CPP-Direktiven formulierten Abhängigkeiten für eines der festgelegten Paare führt zu einem Typfehler, wenn das referenzierte Element, durch die Transformationen des Dateinhaltes entfernt wird. Diese Situation wird von der jeweiligen Annotation eines Elementes beeinflusst. Somit entstehen keine Fehler, wenn beide Bestandteile des Paares gleich annotiert sind beziehungsweise die Bedingung für

den Zugriff der Bedingung des Elementes gleicht oder zusätzlich weiter einschränkt. Werden die beiden Untersuchungsobjekte unterschiedlich markiert, bedeutet dies nicht automatisch einen Typfehler. Für die Anwendung können Beziehungen zwischen Konfigurationsmakros bestehen, wodurch die Markierung eines Zugriffes impliziert, dass die dazugehörige Definition existent ist. Solche Abhängigkeiten werden beispielsweise in einem Merkmalmodell für das Projekt beschrieben. Das Aufzeigen von Typfehlern innerhalb einer Anwendung bedarf der Ermittlung der Präprozessorhierarchie sowie der darin formulierten Bedingungen. Die in *TypeChef* enthaltene Fehleruntersuchung wird im nächsten Abschnitt detailliert beschrieben.

3.5 CSP-Transformation und -Auswertung

Bei der Untersuchung der Referenzpaare auf mögliche Fehlkonfigurationen wird die Präsenz der Elemente ausgewertet, die durch verschiedenartige, hierarchisch strukturierte Präprozessordirektiven formuliert wird. Die Direktiven können dabei eine Vielfalt von Bestandteilen und als Konsequenz daraus eine hohe Komplexität beinhalten (vgl. Kapitel 2.1).

3.5.1 Gestalt der Hierarchieebenen

Die Annotation eines Quellcodebereiches erlaubt nicht nur die Formulierung von komplexen Bedingungen, sondern ebenso eine vielschichtige und umfangreiche Hierarchie. Jedes Auftreten einer `#ifdef`-, `#ifndef`- oder `#if`- Direktive beschreibt den Beginn einer neuen Ebene. Mit Hilfe der CPP-Anweisungen `#elif` und `#else` wird eine Alternative in der aktuellen Ebene erzeugt. Für die Untersuchung der Hierarchie sind das Zusammenspiel der Ebenen und die Direktiven innerhalb einer solchen Ebene zu betrachten. Auf jeder Stufe der Hierarchie ist hinsichtlich der zur Verzweigung genutzten Direktive herauszustellen, wie die verbleibenden Anweisungen zu verarbeiten sind. Die Auswirkungen der vorgelagerten CPP-Anweisungen müssen bei der Konstruktion der Annotationsbedingung während der Referenzanalyse beachtet werden. Für eine Ebene werden folgende Fälle unterschieden:

Verzweigung in `#ifdef`, `#ifndef` oder `#if`: Wird auf einer Ebene in die erste auszuwertende Direktive verzweigt, so wird die formulierte Bedingung unverändert in der Ermittlung eines Hierarchiepfades verarbeitet.

Verzweigung in eine Alternativdirektive: Bei dieser Art von CPP-Anweisungen werden zusätzliche Bedingungen formuliert. Das Zusammenwirken der Direktiven einer Ebene ist abhängig von dem zutreffenden Fall:

`#elif`-Fall: Im `#elif`-Fall wird eine Direktive alternativ erschaffen, sodass alle vorher nicht erfüllten Direktivenbedingungen in negierter Form auszuwerten sind.

`#else`-Fall: Während des Durchlaufens der aktuellen Ebene werden mit der `#if`- und jeder `#elif`-Anweisung neue Bedingungen erzeugt. Ein spezieller Fall

wird durch die Direktive **#else** hervorgerufen, da hierbei keine neue Bedingung definiert, sondern die Negation jeder in der Ebene vorhandenen Direktive ausgedrückt wird.

Im nächsten Abschnitt werden die ebenenspezifischen Informationen ausgewertet und zum Aufbau einer Bedingung genutzt, die den Verbleib eines Quellcodeelementes durch Aussagen über Makros festlegt.

3.5.2 Gesamtbedingung für die Präsenz eines Quellcodeelementes

Für die Beschreibung der Anwesenheit eines Quellcodeelementes wird die umliegende CPP-Hierarchie vollständig untersucht und eine Gesamtaussage konstruiert, die diese Situation mit Hilfe der beteiligten CPP-Anweisungen beschreibt.

Konditionsfunktion für ein Element E:

$$\text{Kondition}(E) = D_i$$

Mit Hilfe der Funktion *Kondition* wird die direkt umschließende CPP-Bedingung für das zu untersuchende Element E ermittelt. Die Direktive stellt die Eingabe für die Präsenzfunktion dar und ist der Ausgangspunkt für die Hierarchieanalyse. Ist das Element nicht annotiert, so liefert die Funktion die Wurzel der CPP-Hierarchie, da der entsprechende Abschnitt immer vorhanden ist.

Zum Aufbau der Gesamtaussage wird die folgende Funktion definiert:

Präsenzfunktion für eine Kondition D_i:

$$\text{Präsenz}(D_i)$$

Der Ausgangspunkt für die Hierarchiebetrachtung ist die CPP-Direktive, die durch D_i ausgedrückt wird und ein fokussiertes Element unmittelbar umschließt. Der Index i repräsentiert die Angabe der zur Verzweigung genutzten Direktive in der aktuell betrachteten Ebene.

Durch eine Konkretisierung der Präsenzfunktion wird der zu durchlaufende Pfad in der Direktivenhierarchie beschrieben und die Beeinflussung der Anwesenheit von E seitens der in den Direktiven hinterlegten Ausdrücke formuliert. Die Funktion *Präsenz* erfasst die vorherrschenden Abhängigkeiten ausgehend von der Direktive D_i, die durch *Kondition*(E) ermittelt wird:

Präsenzfunktion für ein Element E:

$$\text{Präsenz}(D_i) = \begin{cases} 1, & \text{wenn } D_i = \text{Wurzel} \\ \text{Bedingung}(D_i) \wedge \bigwedge_{k=1}^{k < i} (\neg(\text{Bedingung}(D_k))) \wedge, & \text{sonst.} \\ (\text{Präsenz}(\text{U-Ebene}(D_i))) \end{cases}$$

Die Präsenzfunktion setzt sich aus mehreren Teilen und Funktionen zusammen:

Bedingung: Mit Hilfe der Funktion *Bedingung* wird der Inhalt der jeweiligen CPP-Direktive, speziell die darin formulierte Aussage ermittelt und für die Bildung der Gesamtaussage bereitgestellt. Der Spezialfall `#else` muss gesondert betrachtet werden. Die Funktion liefert hierfür den Wahrheitswert 1, da in der Direktive keine zusätzlich zu betrachtende Bedingung festgelegt wird.

Alternativenverarbeitung: Der mittlere Teil der Präsenzfunktion ist die Konjunktion aller Aussagen, in die auf der aktuellen Ebene nicht verzweigt wird. Dieser Ausdruck nimmt den Wert 1 an, sobald keine bereits ausgewerteten Direktiven vorhanden sind. In diesem Fall wird in die erste CPP-Anweisung (`#ifdef`, `#ifndef` oder `#if`) der Ebene verzweigt. Die Verarbeitung der Alternativen nutzt den Index *k* und konjugiert die Negation jeder bis zum Index *i* existierenden Direktive. Die Funktion *Bedingung* entnimmt bei jedem Durchlauf die in der CPP-Anweisung enthaltene Aussage.

Umschließende Ebene (U-Ebene): Die im letzten Gleichungsteil konjunktiv verknüpfte Funktion *U-Ebene* untersucht das Vorhandensein einer weiteren übergeordneten Direktivenebene. Wenn eine solche Ebene existiert, wird die Direktive als neues D_i zurückgeliefert. Durch den Index *i* wird die dort genutzte Verzweigung identifiziert. Wenn keine übergeordnete Ebene existiert, ist die Wurzel der Hierarchie erreicht und wird von der Funktion *U-Ebene* zurückgegeben.

Die Präsenzfunktion wertet die Direktive D_i aus. Falls der übergebene Parameter die Wurzel ist, terminiert das Verfahren und liefert selbst den Wert 1 zurück. Während der Untersuchung der Hierarchie wird dieser Zustand durch die Auswertung der Funktion *U-Ebene* erreicht, die den Rückgabewert Wurzel durch das Fehlen einer übergeordneten Ebene erzeugt.

Die Eingabe für die Prüfung der Präsenz ist die direkt umschließende Direktive. Für ein nicht annotiertes Element existiert die Direktive nicht, sodass die Präsenzfunktion durch *Kondition(E)* mit der Wurzel der Hierarchie initialisiert wird, keine Verarbeitung beginnt und der Wert 1 zurückgeliefert wird.

Die resultierende Gesamtgleichung beschreibt den Pfad durch die Hierarchie. Für zwei Beispiele aus Femto OS wird die Funktion angewendet, um die Aussage über die CPP-Abhängigkeit zu extrahieren.

Quellcodebeispiel aus Femto OS

Das Quellcodebeispiel in Abbildung 3.11 (S. 43) zeigt eine zu untersuchende Direktivenhierarchie, aus der die Bedingung für ein spezifisches Element zu generieren ist. Der Bereich in den Zeilen 5 bis 7 steht in Abhängigkeit zu den konjunktiv verknüpften Aussagen, die sich aus den Direktiven des Pfades (Z. 1, 4) ableiten. Für den Zugriff auf die Variable *cfgGlobSet* in Zeile 4 wird über *Kondition(cfgGlobSet)* die Direktive D_1 (*cfgIntGlobalOnly == cfgTrue*) ermittelt. Im Rahmen der Abhängigkeitsermittlung

```

1  #if (defInterruptStartConstant == cfgFalse)
2  ...
3      #if (cfgIntGlobalOnly == cfgTrue)
4          if ((uiInterruptStart & ((cfgGlobSet | cfgTickSet) ... ) {
5              uiInitCPUStatusRegister |= (1 << portInitGlobalInterruptLoc);
6          }
7      #else
8          if ((uiInterruptStart & (cfgGlobSet & defInitialInterruptGetMask))...) {
9              uiInitCPUStatusRegister |= (1 << portInitGlobalInterruptLoc);
10         }
11     #endif
12     ...
13 #else
14     ...
15 #endif
    
```

Abbildung 3.11: Hierarchie in Präprozessordirektiven (Femto OS)

wird ausgehend von D_1 folgende Bedingung für den Verbleib im Quellcode herausgestellt:

$$\begin{aligned} \text{Präsenz}(D_1) &= (cfgIntGlobalOnly == cfgTrue) \\ &\wedge ((defInterruptStartConstant == cfgFalse) \wedge 1) \end{aligned}$$

Mit dem `#else`-Fall (Z. 8) wird in der zweiten Ebene ein alternativ erreichbarer Abschnitt markiert (Z. 9-11). Auf der aktuellen Ebene wird hierin die Direktive D_2 verwendet. Für den darin hinterlegten Quellcode besteht die Bedingung aus der Aussage, die sich aus der Zeile 1 und somit Ebene 1 ergibt, und aus der dem `#else`-Fall innewohnenden Negation der Bedingung aus Zeile 4. Die Variable `cfgGlobSet` wird innerhalb des Beispielbereiches in Zeile 8 referenziert, wodurch die vorhandenen Präprozessorebenen folgendermaßen abgebildet werden:

$$\begin{aligned} \text{Präsenz}(D_2) &= 1 \wedge \neg(cfgIntGlobalOnly == cfgTrue) \\ &\wedge (defInterruptStartConstant == cfgFalse) \wedge 1 \end{aligned}$$

Die Präsenzfunktion extrahiert die Bedingung für ein Quellcodeelement aus der umliegenden Hierarchie. Die auf diesem Weg gewonnen Informationen bilden die Grundlage für die Typfehleruntersuchungen des nächsten Abschnittes.

3.5.3 Auftreten eines Typfehlers

Die Konfiguration einer Anwendung ist in allen Varianten korrekt, wenn für keine Belegung der verwendeten Makros ein Typfehler im C-Quellcode hervorgerufen wird. Für ein Referenzpaar eines Elementes E resultiert die Notwendigkeit, dass bei einem Zugriff immer die dazugehörige Definition/ Deklaration vorhanden ist. Demnach wird durch eine fehlende Definition ein Typfehler erzeugt. Unter Verwendung der Präsenzfunktion kann die zu untersuchende Situation in einem Ausdruck dargestellt werden, mit dem ein Widerspruch aufgezeigt wird, sobald kein Fehler vorliegt:

$$FM \wedge \neg(\text{Präsenz}(\text{Kondition}(E_{\text{Zugriff}})) \Rightarrow \text{Präsenz}(\text{Kondition}(E_{\text{Erstellen}}))) \Rightarrow \text{Typfehler}$$

Der Ausdruck setzt sich aus den Auswertungen der Präsenzfunktion, speziell der ermittelten Erreichbarkeitsbedingungen eines Elementes, zusammen und wird ausschließlich durch einen Zugriff auf eine nicht anwesende Definition zu einem Wahrheitswert von 1 ausgewertet und letztendlich ein Fehler herausgestellt. Die auf diesem Weg konstruierte Aussage wird durch ein Lösungssystem auf die Erfüllbarkeit überprüft. Wenn das Ergebnis keine Lösung enthält, so tritt kein Fehler auf. In keiner Konfiguration existiert ein Zugriff auf das Element ohne die vorher stattfindende Definition. Als zusätzliche Informationsquelle wird mit FM das Merkmalmodell der zu untersuchenden Anwendung integriert. Darin sind die Beziehungen zwischen den Merkmalen, hier durch die Konfigurationsmakros symbolisiert, festgelegt und werden in die Suche nach Typfehlern einbezogen. Folglich werden die Analyse und falsch bewertete Paare korrigiert.

3.5.4 Überführung in ein Lösungssystem

Die unter Verwendung der Präsenzfunktion ermittelte Gleichung für einen Typfehler kann in einem automatisierbaren Vorgang durch ein geeignetes Lösungssystem überführt werden. Das Lösungssystem stellt für die involvierten Makros heraus, ob eine Belegung im dazugehörigen Wertebereich existiert, für die ein definitionsloser Zugriff auf ein Element entsteht. Im folgenden Abschnitt wird die Überführung in zwei Lösungssysteme erläutert, wobei die Einsatzmöglichkeiten von der Art der verwendeten CPP-Direktiven abhängig sind.

Erfüllbarkeitsproblem

Aus dem Einsatz der Direktiven `#ifdef` sowie `#ifndef` folgt für den umschlossenen Quellcode eine Prüfung auf die Existenz respektive die Abwesenheit einer Definition des in der Kondition verwendeten Makros. Die Verknüpfung der Anweisung mit einem `#else`-Fall bewirkt eine Alternativverzweigung, wenn die Auswertung der vorgelagerten Direktive zu einer Falschaussage führt. Eine solche Verzweigung wird durch den Einschub von `#elif` für zusätzliche Betrachtungen erweitert. Die Definitionsüberprüfung wird in `#if`- und `#elif`-Anweisungen durch das Schlüsselwort *(!)defined* symbolisiert.

Unter der Voraussetzung, dass die Variabilität einer Anwendung ausschließlich durch den Einsatz von definitionsüberprüfenden Direktiven erzeugt wird, kann die Konfigurationsanweisung eines Elementes durch eine aussagenlogische Gleichung beschrieben werden. Hierbei wird ein genutztes Makro als eine Variable des zu konkretisierenden Problems interpretiert, welche die Werte 0 und 1 annehmen kann. Bei der Konstruktion der logischen Formel müssen verschiedene Faktoren beachtet werden. Eine Variable ist negiert zu verwenden, wenn mit der dazugehörigen Direktive die Abwesenheit einer Makrodefinition ausgedrückt wird. Andernfalls bleibt sie unverändert. Die in der Präprozessoranweisung enthaltene Bedingung erfasst eine Mehrfachabhängigkeit durch die Anwesenheit der logischen Verknüpfungen Konjunktion und Disjunktion. Die beiden Verknüpfungen lassen sich direkt in logische Gleichung integrieren, wodurch die ursprüngliche Aussage erhalten bleibt.

Ein fehlerhafter Zugriff wird festgestellt, wenn das Lösungssystem eine Variablenbelegung aufzeigt, die zur Erfüllung der Aussage beiträgt. Als Lösungssystem für das zu

untersuchende Erfüllbarkeitsproblem kann ein SAT-Solver genutzt werden, wenn die An- oder Abwesenheit eines Makros für die Konfiguration des C-Quellcodes genutzt wird. In dem automatisierten Verfahren wird ein Typfehler lokalisiert, wenn eine Variablenbelegung existiert, die die Aussage erfüllt.

Bedingungserfüllungsproblem

Der Präprozessor ermöglicht weit umfangreichere Untersuchungen eines Makros, die nicht mit Hilfe eines SAT-Problems beschrieben werden können. Der CPP erlaubt mit den Direktiven `#if` und `#elif` die Formulierung von Gleichungen, die eine komplexe Auswertung von Makros bezüglich des darin hinterlegten Wertes ermöglichen. Hierbei können verschiedene Verknüpfungsoperatoren zum Einsatz kommen, sodass innerhalb der Anweisung eine Bedingung unter Anderem durch eine arithmetische Gleichung von Makros formuliert werden kann (vgl. Kapitel 2.1). Die Makros repräsentieren in dieser Art von Bedingungen keine logischen Schalter, die die Anwesenheit in einer Variante anzeigen, sondern als ganzzahlige Variablen, die bei der Untersuchung aller Varianten einen beliebigen Wert innerhalb eines festgelegten Bereiches annehmen können. Demnach reicht die Aussagenlogik nicht mehr aus, um ein solches Problem zu beschreiben.

Beispielsweise prüft eine `#if/#elif`-Direktive das Eintreten des folgenden Sachverhaltes: $((X+Y)>7)$. Bei der ursprünglichen Übersetzung durch den CPP werden die speziellen Werte der Makros `X` und `Y` ersetzt, die Summe gebildet und durch den Größenvergleich mit der Ganzzahl 7 zu einem konstanten Ausdruck ausgewertet. Ergibt die Auswertung einen Wahrheitswert $\neq 0$, wird in die CPP-Direktive verzweigt und der darin enthaltene Quellcode bleibt ein Teil der Variante, andernfalls wird er entfernt.

Der gleichzeitige Test aller generierbaren Anwendungen setzt die Betrachtung der Makros `X` und `Y` für den gesamten Wertebereich eines Integers voraus. In Abschnitt 2.1 sind die in den Anweisungen verwendbaren Operatoren aufgeführt. Ein Operator ist als Bedingung zwischen Makros und Konstanten oder weiteren Bedingungen interpretierbar.

Daraus resultiert die Erschaffung eines Bedingungserfüllungsproblems, das für die CPP-Direktive eine Menge von Bedingungen definiert (vgl. Kapitel 2.3.1). Die Aufgabe eines Lösungssystems ist das Aufzeigen einer Variablenbelegung, für die das Problem ein Ergebnis erhält. Ein CSP-Solver nutzt ein Modell des Problems, worin die Gesamtbedingung für einen Typfehler, die Teilbedingungen sowie die Menge der zu betrachtenden Variablen und Konstanten enthalten sind. Die Analyse führt zu einem Typfehler, sobald mindestens eine Variablenbelegung existiert, die für die enthaltene Gesamtbedingung eine Lösung bereitstellt.

3.6 Kombiniertes Analysekonzept

Der Zusammenschluss der in diesem Kapitel gezeigten Techniken erschafft ein Analysesystem für das Auffinden von Typfehlern in C-Präprozessor-konfigurierten Anwendungen. Die einzelnen Arbeitsschritte werden im Werkzeug *TypeChef* kombiniert.

Eine Datei, in der der C-Quellcode mit den Direktiven des CPP verschränkt vorzufinden ist, wird zur Vorbereitung der Referenzanalyse durch den partiellen Präprozessor verarbeitet. Dieser Arbeitsschritt ist ein wichtiger Bestandteil von *TypeChef*, da in dem vorliegenden Dokument alle CPP-Direktiven, die keine bedingte Übersetzung durchführen, vollständig und in der ursprünglichen Funktionalität mit Hilfe eines existierenden C-Präprozessors aufgelöst werden. Folglich muss die spezifische Verarbeitung der Direktive, wie die Expansion von Makros, nicht nachgebildet werden. Das Herauslösen der an der Konfiguration beteiligten Direktiven ist eine vorbereitende Aufgabe des partiellen Präprozessors und verhindert die Transformation des Quellcodes hinsichtlich einer festen Variante. Die betroffenen CPP-Anweisungen werden nachträglich wieder in das Dokument integriert, um die Variabilität wiederherzustellen. Das resultierende Dokument besteht ausschließlich aus C-Code, der im Rahmen der bedingten Übersetzung konfiguriert wird.

Ein Parser für die Programmiersprache C, der zusätzlich CPP-Direktiven analysiert oder um eine solche Funktionalität zu erweitern wird, untersucht die Zusammensetzung des Dokumentes und bildet diese in einem AST ab. In dem Baum sind neben den C-Elementen auch die verwendeten Präprozessoranweisungen enthalten.

Der AST wird genutzt, um eine erweiterte Symboltabelle aufzubauen und Informationen über die Referenzpaare zu ermitteln. Die herausgestellten Daten bilden die Grundlage der weiterführenden Analyse der Konfigurationsabhängigkeit einer Kombination aus Elementdefinition und -zugriff.

Unter Betrachtung der Abhängigkeit kann für ein Referenzpaar gezeigt werden, ob die Kombination der Elemente in jeder konstruierbaren Variante korrekt ist. Ein Typfehler verursacht, wenn der Zugriff ohne das vorgelagerte Erstellen geschieht. Die Fehlerprüfung basiert auf den Bedingungen, die bei der Annotation verwendet werden.

Der abschließende Arbeitsschritt überführt die Bedingungen für einen Typfehler, in Bezug auf die Präsenz des Paares, in ein geeignetes Lösungssystem für Bedingungserfüllungsprobleme, das in einem automatisierten Vorgang bestimmt, inwiefern die Korrektheit in jeder Variante gegeben ist. Dementsprechend können Typfehler, die aus der fehlerhaften Nutzung des CPP resultieren, in der Anwendung identifiziert werden.

3.7 Kapitelzusammenfassung

Im vorliegenden Kapitel wird ein Analysesystem konzeptioniert, mit dessen Hilfe ein Quellcodedokument der Programmiersprache C hinsichtlich der Beeinflussung durch den CPP untersucht werden kann. Zwischen den verschiedenen Elementen in C bestehen Beziehungen, wodurch unterschiedliche Referenzpaare erschaffen werden. Ein solches Paar wird auf das Auftreten eines Typfehlers untersucht, der durch den CPP-Kontext eines beteiligten Elementes verursacht wird.

Grundlegend ist demnach ein Werkzeug zu erschaffen, um die Kombination aus C und CPP strukturell zu untersuchen und einen AST zu erstellen, der die Zusammensetzung des Dokumentes widerspiegelt. Neben den zur Konfiguration genutzten Direktiven stellt der CPP weitere Anweisungen zur Verfügung. Zur Vermeidung der Nachbildung der dahinterliegenden Funktionalität, wie dem Einfügen von gekapselten

Dateiinhalten, wird ein bestehender Präprozessor genutzt. Alle bedingten Anweisungen werden innerhalb von Kommentaren vor der Verarbeitung versteckt, sodass der Quellcode nicht an eine bestimmte Konfiguration angepasst wird.

Die Anpassung des Quellcodes kann auf verschiedenen Ebenen der Granularität auftreten. Im Rahmen der Analyse wird eine Disziplin festgelegt, wodurch die Annotation eines Elementes zwar auf den globalen und lokalen Ebenen des Dokumentes erlaubt ist, jedoch mindestens auf der Anweisungsebene vorkommt. Die Einschränkung erleichtert somit die Analyse der Markierungen, da die korrekte Zusammensetzung eines variabel gestalteten Bereiches nicht zusätzlich zu überprüfen ist.

Der zu generierende AST wird im weiteren Verlauf genutzt, um die geplante Referenzanalyse zu realisieren. Zum Einen werden alle C-Elemente ermittelt, zwischen denen eine Beziehung hergestellt werden kann. Zum Anderen wird die jeweils vorhandene Konfigurationsabhängigkeit extrahiert. Resultierend aus der Herstellung einer Elementbeziehung werden schließlich Referenzpaare gebildet.

Jedes Referenzpaar wird auf das Vorhandensein eines Typfehlers untersucht, der genau dann auftritt, wenn in einer Variante des Projektes der Zugriff auf ein Element trotz fehlender Definition erfolgt. Ein Typfehler definiert auf Grund der in den annotierenden Direktiven formulierbaren Ausdrücke ein CSP, welches durch einen Solver auf Lösbarkeit, speziell eine Makrobelegung, geprüft wird, um einen fehlerhaften Zugriff zu identifizieren. In dem Problem ist für die beiden zu betrachtenden Elemente eine Bedingung enthalten, die die zu durchlaufende CPP-Hierarchie widerspiegelt und somit das Verbleiben im Quellcode festlegt.

Die wiederholte Anwendung von Hierarchietransformation, der Formulierung einer Typfehleraussage als ein CSP und die Prüfung der Erfüllbarkeit für alle zu untersuchenden Paare ist notwendig, um die Fehlerfreiheit für alle generierbaren Konfigurationen sicherzustellen.

KAPITEL 4

Umsetzung der statischen Referenzanalyse

Im vorliegenden Kapitel werden Implementierungsdetails der Phasen von *TypeChef* basierend auf Kapitel 3 beschrieben. Die Umsetzung des partiell angewendeten C-Präprozessors ist Inhalt des Abschnittes 4.1. Hierbei wird erläutert, wie die CPP-Expansion eines Dokumentes ohne dessen Konfiguration realisiert wird. Die vollständig expandierte Datei wird durch einen um CPP-Konstrukte erweiterten Parser in einen AST abgebildet. Hierbei zeigt der Abschnitt 4.2 die Auswahl eines geeigneten Parsergenerators und die Integration eines Regelwerkes für die bedingte Übersetzung. Der erweiterte Parser erstellt einen AST, der die Bestandteile des Quellcodes im jeweiligen Konfigurationskontext hierarchisch organisiert. Die Traversierung der Baumstruktur bildet die Grundlage für die Aufgaben: Disziplinsicherung und der Aufbereitung aller zur Referenzanalyse notwendigen Informationen (vgl. Abschnitt 4.3). Die Annotationen der zusammengestellten Paare werden, wie in Abschnitt 4.4 erklärt, durch *TypeChef* in Bedingungen eines CSP überführt, um mit Hilfe des Lösungssystems Choco auf die Verursachung eines Typfehlers untersucht zu werden (vgl. Abschnitt 4.5). Die verschiedenen Phasen der statischen Referenzanalyse werden in Abschnitt 4.6 zusammengefasst und bilden den Abschluss des Kapitels.

4.1 Partieller C-Präprozessor

Das Ziel des partiellen Präprozessors ist eine kontrollierte Manipulation des zu untersuchenden Quellcodes. Hierbei wird eine Anpassung der Datei vorgenommen, wodurch diese zwar durch den CPP verarbeitet, aber nicht an eine bestimmte Konfiguration angepasst wird (vgl. Abschnitt 3.1).

Eine Besonderheit des CPP ist das Überspringen von auskommentierten Quellcodebereichen, welche unbearbeitet in der Datei verbleiben. Das Ausnutzen dieser Eigenschaft ermöglicht die Realisierung des partiellen Präprozessors, worin die Transformation des Quellcodes ohne die konfigurationsbezogene Anpassung durchgeführt wird. Jedes Vorkommen einer bedingten Anweisung ist somit in einem gesonderten, vorgelagerten Arbeitsschritt in einen Kommentar umzuwandeln.

Für das Herauslösen werden die Dateien in einem gesonderten Vorgang durch Java-basierte Textumformungen modifiziert. Als zusätzliche Vereinfachung wird anfänglich eine mögliche Mehrzeiligkeit der zu betrachtenden CPP-Anweisungen aufgelöst. Der Vereinheitlichung folgt das Auskommentieren der Quellcodezeilen, in denen eine bedingte Übersetzung vorzufinden ist. Analog zum Beispiel aus Abschnitt 3.1 zeigt die Abbildung 4.1 (S. 50) die notwendige Verschiebung des Zeileninhaltes in einen Kommentar (Z. 3, 5). Eine spezielle Anpassung der Kommentartoken wird mittels der

Zeichenkette „@TypeChef“ vorgenommen, wodurch Fehler bei der späteren Reintegration ausgeschlossen werden. Wie bereits in Kapitel 3.1 beschrieben, werden Include Guards als Schutzmechanismus in Header-Files verwendet. *TypeChef* nutzt die Abfrage einer fehlenden Makrodefinition (`#ifndef`) sowie einer direkt folgenden Definition (`#define`) und untersucht den verwendeten Makronamen auf Gleichheit. Stimmt dieser in beiden Fällen überein, wird das Vorhandensein eines Include Guards angenommen und zuzüglich zum schließenden `#endif` nicht auskommentiert.

```
1 #define Tchar unsigned char
2 ...
3 //@TypeChef#if (includeGenLogOs == cfgTrue)
4     void genLogOs(void (*pipe)(Tchar)){...}
5 //@TypeChef#endif
6 ...
```

Abbildung 4.1: Herauslösen der Konfigurationsdirektiven

Die nach den Vorgaben angepassten Dateien werden an den CPP übergeben und durch diesen umstrukturiert. Der GCC wird durch einen Parameter dazu veranlasst, nur die Vorverarbeitung für die C-Dateien auszuführen. Im Rahmen der Umstrukturierung werden die Makros mit dem darin hinterlegten Code ersetzt, die notwendigen Dateien im richtigen Kontext eingefügt und die Funktion aller weiteren Direktiven des CPP standardgerecht umgesetzt (vgl. Abschnitt 3.1). Das Format der Datei und die verwendeten Compiler-spezifischen Elemente werden durch die GNU-C-Analyseeinheit vollständig unterstützt.

Das Dokument wird im nachfolgenden Schritt rücktransformiert, wodurch die Elemente in die anfängliche Abhängigkeit der variablen Gestaltung zurückversetzt werden. Hierbei sind alle auskommentierten Anweisungen zu ermitteln und die vorgeschobenen Kommentartoken zu entfernen. Der in dem Kommentar enthaltene Zusatz bewirkt, dass ausschließlich die im Vorfeld auskommentierten Zeilen reintegriert werden.

Das erläuterte Verfahren löst eine Vielzahl an schwierigen Problemen. Jedoch können in einem C-Projekt Quellcodekonstrukte existieren, die die bedingte Übersetzung mit weiteren CPP-Direktiven kombinieren. An dieser Stelle wird eine Grenze des partiellen Präprozessors aufgezeigt. Eine detaillierte Analyse dieser Grenzen wird in Abschnitt 6.2 vorgenommen.

Eine transformierte, aber nicht konfigurierte Datei ist das Ergebnis des beschriebenen Vorganges und bildet die Grundlage für die Analyse von Referenzpaaren unter dem Gesichtspunkt, dass Typfehler durch die Beeinflussung des CPP hervorgerufen werden.

4.2 Präprozessorerweiterung des C-Parsers

An das zu realisierende Analysesystem werden verschiedene Anforderungen gestellt. Eine grundlegende Eigenschaft ist die Fähigkeit, den C-Quellcode in einem bestehenden Projekt zu untersuchen. Die Sprache C hat Fehlerkorrekturen und Funktionserweiterungen erhalten, die in verschiedenen Standards festgelegt sind [Str02]. Eine Einführung von Neuerungen und Verbesserungen ist in C von der ersten Version, die

KERNIGHAN und RITCHIE [KR88] definierten, über die Standardisierung als ANSI-C bis zur Version C99 geschehen. Wird seitens des Analysewerkzeuges ein solcher Standard vollständig unterstützt, können entsprechende C-Projekte untersucht werden.

Die zweite Anforderung ist auf die Auswertung des Quellcodes unter Betrachtung der Variabilität erzeugenden Präprozessordirektiven ausgerichtet. Wenn die Anweisungen des CPP nicht bereits vom System analysiert werden können, bedarf es einer Möglichkeit diese Art der Annotation zu integrieren.

Eine weitere Aufgabe betrifft das Resultat der Untersuchung. Hierbei ist die automatische Erstellung eines AST gefordert, der die Struktur des Inhaltes eines C-Dokumentes widerspiegelt. Die Repräsentationsform ist zusätzlich auf das Vorhandensein von CPP-Direktiven auszurichten, um die notwendigen Informationen für die weiterführenden Untersuchungen im Kontext des Präprozessors bereitzustellen.

Die Analyse der Zusammensetzung einer Quellcodedatei ist grundlegend durch einen Parser zu realisieren. Für die Erschaffung eines Parsers werden verschiedene, freie Generatoren zur Verfügung gestellt, wodurch eine manuelle Durchführung der Entwicklung nicht notwendig ist. In einem automatisierten Vorgang wird die Parserkomponente, auf der Basis einer Definition der zu untersuchenden Programmiersprache, generiert. Für die Aufgabe steht eine Vielzahl von Generatoren zur Verfügung. Die Menge wird dahingehend begrenzt, dass die Zielsprache für *TypeChef* auf C# oder Java festgelegt wird. Im weiteren Verlauf des Abschnittes werden die Systeme: Grammatica¹, ANTLR², GOLD Parser Builder³ und JavaCC⁴ auf die Eignung für das in *TypeChef* verfolgte Vorhaben untersucht.

Aus den betrachteten Parsergeneratoren ist jener auszuwählen, der den größten Anwendungsbereich abdeckt. Für den Generator Grammatica wird keine C-Grammatik bereitgestellt, sodass die Entwicklung einer solchen Sprachbeschreibung notwendig ist [CR05]. Die Anforderung bezüglich des Verständnisses wird von allen weiteren Systemen eingehalten. Der allgemeine Standard ist ANSI-C, wobei in ANTLR mit GNU-C eine höhere Version untersucht werden kann.

Alle Generatoren arbeiten auf der Basis von Grammatiken und erlauben das Hinzufügen von Regeln, die das Parsen von CPP-Direktiven ermöglichen. Die Integration neuer Grammatikelemente wird durch die sachgerechte Verwendung der Definitionssprache begrenzt, um eine Unterstützung der Präprozessoranweisungen in den zulässigen Bereichen des Quellcodes zu erlauben.

Abschließend sind die Werkzeuge in Hinsicht auf das Resultat der Quellcodeanalyse zu betrachten. ANTLR beinhaltet einen Mechanismus, der den AST eines Dokumentes automatisch erzeugt. Des Weiteren kann in diesem Vorgang, unter Verwendung einer speziellen Sprache, die Modifikation und Restrukturierung des Baumes erfolgen. Der Generator GOLD erstellt innerhalb des Durchlaufes einen Parserbaum, in welchem bei einer CPP-Erweiterung der Grammatik, die dazugehörigen Elemente wiederzufinden sind. Die Darstellung der Quellcodestruktur kann jedoch nicht modifiziert werden, wodurch keine zusätzlichen Informationen zur Erleichterung der Analyse bereitgestellt

¹ <http://grammatica.percederberg.net/index.html>

² <http://www.antlr.org/>

³ <http://www.devincook.com/goldparser/index.htm>

⁴ <https://javacc.dev.java.net/>

werden können. In der Grammatik des Generators JavaCC ist ein solcher Mechanismus nicht standardmäßig enthalten. Demnach muss das Erstellen eines AST nachträglich mittels einer Bibliothek, wie JJTree⁵, ergänzt werden.

Die Wahl des zu nutzenden Parsergenerators beruht auf der Erfüllung der gestellten Anforderungen. Ohne weiteren Mehraufwand werden alle Anforderungen durch GOLD und ANTLR erfüllt. Für die folgenden Quellcodeuntersuchungen wird jedoch ANTLR als Generator gewählt, da die existierende Grammatikerweiterung die Analyse eines höheren Standards ermöglicht. Ein weiterer Vorteil ist die Beeinflussbarkeit der AST-Generierung, woraus eine Erleichterung der Analyse resultiert. Der Generator ANTLR wird im Folgenden näher beleuchtet.

4.2.1 ANother Tool for Language Recognition - ANTLR

Als Grundlage für die Untersuchungen wird der freie, objektorientierte Parsergenerator *ANother Tool for Language Recognition (ANTLR)* genutzt. ANTLR dient der Entwicklung von verschiedenen Werkzeugen (Parser, Lexer, TreeParser), die die Analyse von Quellcodedateien für LL(k)-Grammatiken ermöglichen [Par07, LZW⁺08]. Eine LL(k)-Grammatik ist eine spezielle kontextfreie Form, bei der k die Anzahl von vorauszuschauenden Symbolen angibt, um einen Ableitungsschritt eindeutig zu bestimmen. Der Parsergenerator bietet, im Gegensatz zu anderen Vertretern, verschiedene Vorteile, die die Werkzeugentwicklung vereinfachen und den resultierenden Funktionsumfang erhöhen [PQ95, BP08, Bre09]. Eine ANTLR-Grammatik beinhaltet Regeln zur Definition von Lexer und Parser für eine Programmiersprache. Der sprachspezifische Lexer zerlegt eine Folge von Zeichen in *Token*, wodurch Sprachbestandteile, wie Schlüsselwörter oder Bezeichner, identifiziert werden. Die Token werden in der syntaktischen Analyse genutzt und je nach Tokenkombination in passende Regeln eingeordnet.

ANTLR unterstützt eine Vielzahl von Zielsprachen, wie Java, C# oder Python, in der die Komponenten Lexer und Parser in einem automatischen Verarbeitungsschritt aus der Grammatik generiert werden. Innerhalb der Grammatik kann eine zusätzliche Semantik in der verwendeten Zielsprache hinterlegt werden, um spezifische Funktionalitäten zu realisieren [Par07].

Im Zuge der lexikalischen und syntaktischen Analyse wird ein abstrakter Syntaxbaum für die Quellcodedatei erstellt. Die Generierung wird vollständig von ANTLR übernommen und in den Parser integriert. Hierbei erlaubt der Generator wiederum spezielle Anpassungen, die eine Strukturveränderung im AST hervorrufen oder eine Manipulation durch das Hinzufügen von Pseudoelementen ermöglichen [Par08].

4.2.2 C-Grammatik und Präprozessorvariabilität

Für den Parsergenerator ANTLR wird ein Framework zur Quellcodetransformation zur Verfügung gestellt, das die Analyse der Programmiersprache C erlaubt. Das Paket ist auf die Zielsprache Java zugeschnitten und enthält eine Grammatik zur Generierung von Lexer und Parser des Standards GNU-C.

⁵ <https://javacc.dev.java.net/doc/JJTree.html>

Die vorhandene Grammatik ignoriert in der Grundform alle mit der Raute beginnenden Zeilen. Dementsprechend beinhaltet sie keine Untersuchungsregeln für den CPP beziehungsweise die kombinierte und verschränkte Verwendung von Präprozessor und C. Für die Verarbeitung solcher Sprachkonstrukte ist eine Erweiterung des bestehenden Regelwerkes notwendig.

Definition der CPP-Regeln

Die Anwendung des CPP innerhalb von C-Dateien bewirkt, neben dem Einfügen anderer Inhalte und der Textersetzung durch Makros, die Durchführung einer bedingten Übersetzung (vgl. Abschnitt 2.1). Bei dieser speziellen Anpassung wird der umschlossene Quellcode durch die Auswertung einer CPP-Kondition transformiert.

Zur Integration der bedingten Übersetzung in die bestehende Grammatik wird eine neue Hauptregel *CPPLine* erzeugt. Auf der Basis dieses übergeordneten Elementes wird in die spezielle Weiterverarbeitung der Direktiven verzweigt (vgl. Abbildung 4.2).

```

1 CPPLine
2   : POUNDS! (conditionalCompilation)
3   ;

```

Abbildung 4.2: Hauptregel für die Verzweigung in die CPP-Direktiven

Jede Quellcodezeile, die eine Präprozessoranweisung beinhaltet, beginnt mit einer signifikanten Charakteristik, dem `#` (POUNDS, Z. 2), auf welche ein Schlüsselwort, wie z. B. `ifdef`, folgt. An dieser Stelle ist eine weitere Verzweigung notwendig, um die unterschiedlichen CPP-Direktiven in gesonderte Regeln zu kapseln, abzuhandeln und spezielle Funktionalität zu hinterlegen. Für die bedingte Übersetzung des Präprozessors wird über die Regel *conditionalCompilation* ein separater Analysestrang festgelegt. Für jede verfügbare Anweisung wird die mögliche Zusammensetzung definiert, um alle Bestandteile des Ausdrucks zu erfassen und parsen zu können.

In Abbildung 4.3 (S. 54) ist die Teilregel zum Erkennen der Direktive `#ifdef` dargestellt. Das vorangestellte Zeichen `#` wird bereits auf der darüberliegenden Ebene erkannt und muss in dieser Regel nicht mehr verarbeitet werden. Die Zeichenkette „`ifdef`“ in Zeile 2 legt den Typ der Direktive fest.

Der Aufbau des AST wird mittels einer ANTLR-internen Spezifikationssprache manipuliert (Z. 6, 10). Hierbei werden an Stelle der Knoten „`ifdef`“ und `ID` durch *PPifdef* eine neue Wurzel und durch *PPdefined* ein darunter angehängter Kindknoten erzeugt. Der eigentliche Bezeichner wird der Hierarchie wiederum als Kind untergeordnet.

Die neuen Bauelemente *PPifdef* und *PPdefined* gehören zu der durchgeführten Grammatikerweiterung und werden dieser als Pseudoknoten hinzugefügt. Die Zuhilfenahme solcher Ersatzknoten ermöglicht die Vereinheitlichung der zu untersuchenden Strukturen durch die gezielte Anpassung des AST und bewirkt letztendlich eine Erleichterung für spätere Analysen.

Die CPP-Direktive `#ifndef` prüft während der Transformation auf das Fehlen einer Makrodefinition. Die Grammatikregel setzt sich aus der in der Abbildung 4.4 (S. 54) dargestellten Struktur zusammen. Die Zeichenkette „`ifndef`“ wird an dieser Stelle als

```

1 conditionalCompilation
2   : ("ifdef"!
3     (
4       (id1:ID!)
5       {
6         ## = #( #[PPifdef, ""], #[PPdefined, ""], id1));
7       }
8       | (LPAREN! id2:ID! RPAREN!)
9       {
10        ## = #( #[PPifdef, ""], #[PPdefined, ""], id2));
11      }
12     )
13     ...
14   ;

```

Abbildung 4.3: Grammatikdefinition für die Direktive `#ifdef`

Indikator für die Verzweigung erwartet. Die weiterführende Abhandlung modifiziert den erzeugten AST, sodass eine Hierarchie aus den Knoten *PPifndef*, *PPnotdefined* und der ermittelten ID erzeugt wird.

```

1 conditionalCompilation
2   : ...
3     ("ifndef"!
4     (
5       (id1:ID!)
6       {
7         ## = #( #[PPifndef, ""], #[PPnotdefined, ""], id1));
8       }
9       | (LPAREN! id2:ID! RPAREN!)
10      {
11        ## = #( #[PPifndef, ""], #[PPnotdefined, ""], id2));
12      }
13     )
14     ...
15   ;

```

Abbildung 4.4: Grammatikdefinition für die Direktive `#ifndef`

Darüber hinaus wird die zulässige Zusammensetzung der bedingten Direktiven `#if`, `#elif` und `#else` festgelegt. In der Abbildung 4.5 (S. 55) sind die erlaubten Teilausdrücke dargestellt. Mit Hilfe der Ausdrücke in den Zeilen 3 und 5 wird auf das Eintreten einer Bedingung geprüft. In den beiden Regeln wird über die Produktion *PPIfExpression* eine Gleichung von Makros untersucht. Unterhalb der Regel *PPIfExpression* werden hierbei alle nutzbaren Operatoren und Operanden abgebildet (vgl. Kapitel 2.1).

Die Operatorenhierarchie ist aus der bestehenden Grammatik adaptiert und strukturiert die mögliche Gleichungszusammensetzung. Das entstehende Regelwerk beschreibt die Reihenfolge der Auswertung der verwendbaren Operatoren, wodurch eine logische und-/ oder-Beziehung erst nach den relationalen und den darin enthaltenen, arithmetischen Verknüpfungen aufgelöst wird.

Die Analyse des Schlüsselwortes „(!)defined“ dient der Prüfung auf An- oder Abwesenheit einer Makrodefinition. Die Grammatik beeinflusst die Konstruktion des Baumes durch die Integration eines der beiden Pseudoknoten *PPdefined* oder *PPnotde-*

```

1 conditionalCompilation
2   :
3   ...
4   ("if"! PPIfExpression)
5   {
6     ## = #( #[PPif, ""], ##);
7   }
8   | ("elif"! PPIfExpression)
9   {
10    ## = #( #[PPelif, ""], ##);
11  }
12  | ("else"! )
13  {
14    ## = #( #[PPelse, ""], ##);
15  }
16  ...
17  ;

```

Abbildung 4.5: Grammatikdefinition für die Direktiven `#if`, `#elif`, `#else`

*fin*ed. Daraus resultiert für die spätere Untersuchung der Bedingungen eine vereinheitlichte Abhandlung. Die Anwendung dieser Restrukturierung erschafft für die zwei äquivalent anwendbaren Direktiven zur Abfrage einer Makrodefinition eine identische Teilbaumstruktur.

Die Abbildung 4.6 zeigt die syntaktische Verarbeitung der Direktive `#endif`. Die Anweisung markiert das Ende eines konfigurierten Quellcodebereiches. In der Parserregel wird durch das Auftreten der Zeichenkette „endif“ der Ersatzknoten *PPendif* im resultierenden AST erschaffen.

```

1 conditionalCompilation
2   :
3   ...
4   | ("endif"! )
5   {
6     ## = #( #[PPendif, ""], ##);
7   }
8   ;

```

Abbildung 4.6: Grammatikdefinition für die Direktive `#endif`

In der Hauptregel *CPPLine* werden die Präprozessordirektiven hinsichtlich der erlaubten Zusammensetzung beschrieben. Für die durchzuführende Untersuchung werden die Annotationen als diszipliniert angenommen (vgl. Kapitel 3.2). Dementsprechend erfolgt die Erweiterung der Grammatik an die Disziplin erhaltenden Positionen, um in die *CPPLine* zu verzweigen und an Stelle einer Präprozessordirektive zu erlauben. Auf der globalen Ebene ist die neue Hauptregel als Alternative zur Behandlung von Deklarationen sowie der Definition von Funktionen und benutzerdefinierten Typen einzufügen. Folglich können die hier auftretenden Elemente annotiert werden. Weiterhin ist der Einsatz des CPP in den lokalen Bereichen des Quellcodes möglich. Die *CPPLine* ist eine zusätzliche Regel innerhalb eines gekapselten Blockes, der einer Funktions-, Struktur-/ Union- beziehungsweise Aufzählungstypdefinition zugeordnet wird. Daraus resultiert die Zulässigkeit von Annotationen, die in der Deklarations-

und Anweisungsliste einer Funktion sowie für die Komponenten einer Struktur, Union oder Aufzählung eine variable Anpassung hervorrufen.

Das Hinzufügen der neuen Regeln bewirkt eine Erweiterung des Analyseumfanges der vorhandenen ANTLR-Grammatik um CPP-Anweisungen. Der durch den Parser erstellte AST wird mit den dazugehörigen Informationen versehen, sodass für den konfigurierten Quellcode untersucht werden kann, unter welchen Bedingungen die Transformation des CPP greift.

4.2.3 Erweiterter Parser

Die verwendete Grammatik erhält das im Zuge der vorliegenden Arbeit entwickelte Regelwerk, um die bedingten Anweisungen des CPP abzudecken. Daraus werden durch ANTLR die Java-Dateien generiert, worin die Funktionalität der Werkzeuge Lexer und Parser enthalten ist.

Der Lexer zerlegt eine vorliegende Datei anhand von lexikalischen Mustern und vollzieht die Zuordnung des Inhaltes zu einer Menge von Token, die die kleinste Sinngebende Einheit für die Programmiersprache C definieren [CT04].

Im Parser werden die Token untersucht und unter Verwendung der enthaltenen Regeln in verschiedenartige Kombinationen eingeordnet, die die erlaubten Quellcodeelemente, wie eine Funktionsdefinition, repräsentieren [CT04]. Eine Tokenmenge ist genau dann zulässig, wenn sie vom Parser fehlerfrei eingeordnet wird.

Sobald Lexer und Parser eine Datei vollständig bearbeiten, sind die Bestandteile und deren Benutzung im jeweiligen Kontext valide. In Bezug auf die erweiterte Grammatik ist die Korrektheit gegeben, wenn im Quellcode ausschließlich Annotationen vorhanden sind, die gemäß der Vorgaben als diszipliniert erkannt werden.

Das Analyseresultat wird in einem AST abgelegt, wodurch die Quellcodeelemente und die CPP-Direktiven hierarchisch organisiert werden. Im Folgenden werden die Traversierung des Baumes und die Aufbereitung der Informationen für die Referenzpaare erläutert.

4.3 Informationsaufbereitung für Referenzpaare

Die Untersuchung der Referenzpaare in *TypeChef* wird in Form einer Java-Anwendung realisiert. In der ersten Phase wird das Quellcodedokument unter Zuhilfenahme des partiellen Präprozessors vorbereitet, sodass ausschließlich der konfigurationsabhängige C-Quellcode in das Parsersystem übergeben wird. Innerhalb der Konzeptbeschreibung schließt sich eine Testphase an, um die Einhaltung der festgelegten Disziplin zu sichern. Der Test wird jedoch nicht in Form einer einzelnen *TypeChef*-Phase realisiert, sondern ist in den erweiterten Parser integriert und wird zusätzlich durch eine spezielle Untersuchung des AST abgesichert. Das genaue Vorgehen wird in dem folgenden Abschnitt im Zusammenhang mit dem grundsätzlichen Durchlaufen des AST beschrieben.

4.3.1 Traversierung des AST

Nach dem Parser-Schritt wird die Traversierung des generierten AST vollzogen und somit die Sammlung der benötigten Informationen ermöglicht. Die auftretenden Beziehungen zwischen den hierarchisch organisierten Baumbestandteilen werden für jeden der Knoten durch die Zuordnung eines Kind- sowie eines Geschwisterknotens dargestellt, wenn vom aktuellen Element in einen tieferliegenden beziehungsweise auf ebendieser Ebene direkt nebenliegenden Teilbaum verzweigt werden kann.

Der AST für eine vom erweiterten Parser vollständig verarbeitete Datei wird durch einen Besucher (engl. Visitor) abgearbeitet. In jedem Visitor-Schritt besteht die Möglichkeit, den aktuellen Knoten auf dessen Eigenschaften zu untersuchen. Für die Analysen wird durch eine Fallunterscheidung über den Typ des Knotens herausgestellt, ob eine spezielle Abarbeitung angestoßen werden muss. An dieser Stelle werden die Pseudoelemente, die während des Parserdurchlaufes als Analyse erleichternde Indikatoren in den AST integriert werden, bei der Entscheidungsfindung genutzt. Die nächsten Abschnitte beschreiben die Verwendung der erwähnten Zusatzknoten, um alle notwendigen Daten für die Direktivenhierarchie, das Erschaffen und den Zugriff zur Bildung der Referenzpaare zu extrahieren und für die Korrektheitsuntersuchungen bereitzustellen.

Eine abgewandelte Form des beschriebenen Besuchers ist in *TypeChef* integriert und stellt eine die Disziplin verletzende Annotation fest, insofern diese nicht bereits im erweiterten Parser zu Fehlern führen.

4.3.2 Disziplinierte Annotationen

Die Festlegung des Fokus auf eine disziplinierte Ausführung der Annotationen erleichtert die Untersuchungen der Quellcodebestandteile. Die Grammatik wird an verschiedenen Punkten erweitert, um im Parser in die Verarbeitungsregel für die CPP-Direktiven zu verzweigen sowie die mögliche Zusammensetzung abzuarbeiten. Die Positionen für die Erweiterungen sind auf die Unterstützung von disziplinierten Markierungen ausgerichtet. Folglich werden die Präprozessoranweisungen ausschließlich als Alternativen zu den markierbaren Quellcodebestandteilen erlaubt, sodass auf der Ebene des Annotationsinhaltes neben den ursprünglichen Elementen eine gleichrangige platzierte Direktive zulässig ist. Der erweiterte Parser folgt während der syntaktischen Analyse den um die bedingten Direktiven erweiterten Mustern. Sobald der Parser eine CPP-Bedingung nicht einordnen kann, wird die Abarbeitung unterbrochen und das betroffene Quellcodefragment dokumentiert.

Der Grammatikzusatz benötigt jedoch einen weiteren Mechanismus, um die Disziplin vollständig zu gewährleisten. Der Parser erkennt eine vorhandene Direktive, entscheidet dabei allerdings nicht über die korrekte Zuordnung einer Weiterverzweigung beziehungsweise den Abschluss zur jeweils öffnenden Anweisung.

In einer abgewandelten Form des Visitors wird der erstellte AST in Hinblick auf die Einhaltung der Disziplin untersucht. Den Betrachtungen liegt die Tiefe des Baumes zu Grunde und gilt als Entscheidungskriterium für die Korrektheit eines von der Konfiguration abhängigen Blockes. Das Auftreten einer öffnenden Direktive bewirkt die Speicherung der aktuellen Baumtiefe. Für jede alternative und schließende Anwei-

sung ist die letzte öffnende Anweisung zu ermitteln. Über den Vergleich des jeweiligen Tiefenwertes wird eine disziplinierte Annotation herausgestellt, wenn die Ebenen übereinstimmen. Der Wertevergleich zeigt eine Verletzung der Disziplin auf, wenn ein Unterschied zwischen den Tiefen besteht.

Der Disziplin-Visitor durchläuft den AST und überprüft die vorhandenen Direktiven in Abhängigkeit von der Ebene, in der sie eingesetzt werden. Eine nicht disziplinierte Annotation wird mit einer Fehlermeldung angezeigt, wobei die Art der Präprozessoranweisung und die Zeile, in der die Anweisung vorzufinden ist, ausgegeben werden. Aufbauend auf der Anzeige müssen notwendige, manuelle Änderungen am Quellcode vorgenommen werden, wodurch das Dokument in eine disziplinierte Form überführt wird.

Sobald der AST ohne eine Beanstandung durch den Visitor traversiert wird, ist die geforderte Disziplin erfüllt. Die im weiteren Verlauf des Kapitels erläuterten Analysen nutzen den überprüften Syntaxbaum, um daraus alle notwendigen Informationen zu entnehmen.

4.3.3 Ermittlung der Direktivenhierarchie

Die Untersuchung der Referenzen stützt sich auf Fehler, die durch das Vorhandensein von Konfigurationsanweisungen entstehen. Die Direktiven sind im AST gleichrangig zu den markierten Elementen hinterlegt und kennzeichnen die Zugehörigkeit zu einer speziellen Bedingung. In Kapitel 3.5 wird für ein potenzielles Untersuchungsobjekt die Grundidee zur Ermittlung der Gesamtbedingung skizziert. Hierbei wird ein Pfad von Direktiven aus den vorhandenen Ebenen der Präprozessorhierarchie zusammengestellt, um das Erreichen des entsprechenden Elementes auszudrücken.

Die Gesamtbedingung wird im Analysesystem als ein aktueller Zustand registriert. Während der Traversierung des AST wird bei jedem Auftreten einer CPP-Direktive die Anpassung des Zustandes hervorgerufen. Dabei ist auf die drei folgenden Veränderungen zu reagieren:

Erschaffen einer neuen Ebene: Das Auftreten einer der Direktiven `#ifdef`, `#ifndef` oder `#if` legt eine zusätzlich zu betrachtende Präprozessorebene fest, die den aktuellen Zustand durch das Hinzufügen eines weiteren Bedingungsstranges verändert.

Erweiterung der Ebene: Die Anweisungen `#elif` und `#else` sind Alternativverzweigungen innerhalb der momentanen Ebene und werden zur Formulierung einer Zusatzbedingung genutzt, wodurch die Notwendigkeit der Negierung aller vorhergehenden Bedingungen der Ebene gekennzeichnet wird.

Abschluss der Ebene: Mit Hilfe der Anweisung `#endif` wird das Ende einer Ebene definiert. Der aktuelle Konfigurationszustand wird durch das Entfernen der dazugehörigen Bedingungen modifiziert.

Die Verfolgung der Bedingungshierarchie ist ein gleichzeitig zur Ermittlung der Referenzinformationen stattfindender Prozess. Innerhalb einer globalen Liste wird für

jede Stufe der Direktiventiefe ein Datenobjekt erschaffen. Darin werden die vorherrschenden Ebenenbedingungen in ihrer Zusammensetzung, insbesondere der AST der CPP-Direktive, gekapselt (vgl. Kapitel 3.5.1). Jedem nicht annotierten Quellcodebereich wird eine leere Liste zugeordnet und als von der Konfiguration unabhängig gekennzeichnet.

Die auf dem beschriebenen Weg entstehende Informationsquelle wird für die folgenden Untersuchungen benötigt. Darin ist der aktuelle Kontext bezüglich des jeweils geltenden Direktivenpfades hinterlegt. Ein C-Dokument beinhaltet in Beziehung stehende Elemente, für die im weiteren Verlauf des Abschnittes alle Informationen aus dem AST extrahiert werden. Die Informationen werden in einer speziellen Datenstruktur, für die Ermittlung von zusammengehörigen Analyseobjekten, verwaltet.

4.3.4 Befüllung der Symboltabelle

Die Phase der Referenzermittlung nutzt die im AST hinterlegten Informationen und baut die erweiterte Symboltabelle auf. Der verwendete Visitor durchläuft den Baum und entscheidet anhand der Art des jeweiligen Knotens, inwiefern eine spezielle Abarbeitung notwendig ist. Die in *TypeChef* genutzte Grammatik verbessert die Auffindbarkeit der elementspezifischen Muster mit Hilfe von Pseudoknoten, die in den AST integriert werden. Unter Verwendung von Bäumen für Quellcodebeispiele aus Femto OS wird die generelle Arbeitsweise von *TypeChef* während der Ermittlung der Referenzen aufgezeigt. Der jeweils extrahierte AST wird in der internen Repräsentation von *TypeChef* dargestellt. Jede Zeile dieser Abbildung zeigt einen AST-Knoten, die sich aus den Informationen: dem Textinhalt des Knotens, der Nummer und der Bezeichnung des erkannten Tokens zusammen. Die durch *TypeChef* entnommenen Informationen werden anhand der Bezeichnung zu elementspezifischen Mustern zugeordnet, wobei wichtige Elemente, wie der Name der definierten Funktion, auf dem hinterlegten Text aufbauen.

Für den in Abbildung 4.7 (S. 60) dargestellten Auszug aus Femto OS beinhaltet die Abbildung 4.8 (S. 60) den AST der Funktionsdefinition und der darin enthaltenen Quellcodeelemente. Der charakteristische Einstiegspunkt für den Visitor ist der Pseudoknoten *NFunctionDef* (Z. 1). In den Zeilen 2 bis 13 wird die Signatur in einem Teilbaum hinterlegt. *TypeChef* analysiert die Struktur, wobei durch die *ID* (Z. 5) und eine direkt folgende *NParameterTypeList* (Z. 6) der eindeutige Bezeichner festgelegt wird. Anhand der in Zeile 6 beginnenden Parameterliste können zwei wichtige Punkte der Referenzermittlung erläutert werden. Die Grammatik unterstützt das Auffinden der Nutzung einer Typdefinition durch den Knoten *NTypedefName*. Über die *ID* wird das Element benannt, sodass eine Referenzbeziehung zur Definition hergestellt und in die Typfehlerprüfung übergeben werden muss. Mit Hilfe des Synonyms wird eine Variable als Parameter deklariert, die in die erweiterte Symboltabelle eingetragen wird und als Gültigkeitsbereich die aktuell verarbeitete Funktion erhalten muss.

Der nächste abgebildete Pseudoknoten *NExpr* (Z. 15) ist der Indikator für den Beginn eines Ausdrucks. Die darunterliegenden Elemente werden durch die Angabe einer *ID* als Variablennutzung registriert (Z. 17, 21, 22) und lösen die Bildung von Referenzpaaren zur passenden Deklaration des Elementes aus. Im rechten Teil des Beispiels ist

```

1 static void privShowError(Tuint08 uiMessage, ...)
2 {
3 ...
4     uiBareMessage = (uiMessage & errMessageGetMask);
5 ...
6     if (!bReturn)
7     {
8         privSetStack(&xOS.StackOS[OSstackInit]);
9     }
10 }

```

Abbildung 4.7: Quellcodeauszug zum AST Beispiel 1 (Femto OS)

<pre> 1 : 112 : NFunctionDef 2 static : 16 : "static" 3 void : 18 : "void" 4 : 100 : NDeclarator 5 privShowError : 27 : ID 6 (: 111 : NParameterTypeList 7 : 114 : NParameterDeclaration 8 : 98 : NTypedefName 9 Tuint08 : 27 : ID 10 : 100 : NDeclarator 11 uiMessage : 27 : ID 12 ... 13) : 33 : RPAREN 14 { : 113 : NCompoundStatement 15 ... 16 : 165 : NExpr 17 = : 30 : ASSIGN 18 uiBareMessage : 27 : ID 19 (: 105 : NExpressionGroup 20 : 165 : NExpr 21 & : 64 : BAND 22 uiMessage : 27 : ID 23 errMessageGetMask : 27 : ID 24) : 33 : RPAREN 25 ... </pre>	<pre> 26 if : 46 : "if" 27 : 165 : NExpr 28 : 116 : NUnaryExpr 29 ! : 81 : LNOT 30 bReturn : 27 : ID 31 { : 113 : NCompoundStatement 32 : 109 : NStatementExpr 33 : 165 : NExpr 34 : 118 : NPostfixExpr 35 privSetStack : 27 : ID 36 (: 106 : NFunctionCallArgs 37 : 116 : NUnaryExpr 38 & : 64 : BAND 39 : 118 : NPostfixExpr 40 xOS : 27 : ID 41 : 168 : NStructCompCall 42 . : 83 : DOT 43 StackOS : 27 : ID 44 [: 34 : LBRACKET 45 : 165 : NExpr 46 OSstackInit : 27 : ID 47] : 35 : RBRACKET 48) : 33 : RPAREN 49 } : 8 : RCURLY 50 } : 8 : RCURLY </pre>
--	--

Abbildung 4.8: AST Beispiel 1 (Femto OS)

eine weitere *NExpr*, hier zur Formulierung der Bedingung einer Steuerstruktur, dargestellt (Z. 27-30).

Abschließend zeigen die Zeilen 32 bis 47 einen komplexeren Aufruf, der durch *TypeChef* hinsichtlich der Zusammensetzung zerlegt werden muss. Die *ID* (Z. 34) wird auf einen nachfolgenden Knoten des Typs *NFunctionCallArgs* getestet, da dieses Muster einen Funktionsaufruf beschreibt. Ausgehend von dem Bezeichner ist ein Paar mit der passenden Definition/ Deklaration herzustellen und in die Fehlerprüfung zu überführen. Ab der Zeile 36 wird die Struktur der übergebenen Parameter aufgeschlüsselt.

Die *ID* *xOS* in Zeile 40 ist ein normaler Variablenzugriff. Im Anschluss daran impliziert der Knoten *NStructCompCall* (Z. 41) die Nutzung einer Struktur-/ Union-Komponente mit dem Bezeichner *StackOS*. An dieser Stelle muss auf eine Limitierung der Analyse hingewiesen werden. *TypeChef* führt keine detaillierte Typprüfung durch, um für eine genutzte Komponente die dazugehörige Struktur/ Union zu ermitteln. Folglich wird eine Referenzbeziehung nur dann hinsichtlich der CPP-Bedingungen untersucht, wenn die Symboltabelle eine eindeutig zuzuordnende Komponente enthält. Im Verlauf

des Baumes wird wiederum eine *NExpr* gefunden und in der bereits beschriebenen Vorgehensweise verarbeitet.

Während der Verarbeitung dieses Beispiels wird die Symboltabelle um neue Einträge ergänzt. Der AST beinhaltet ausschließlich zwei erstellende Operationen. Folglich werden eine Funktionsdefinition für *privShowError* und für den Parameter *uiMessage* die Deklaration einer Variablen als Einträge für die Tabelle 4.1 erzeugt. Die Punkte am Anfang der Tabelle zeigen die Existenz weiteren Einträgen an, da alle vor diesem Quellcodebeispiel befindlichen und ein Element erschaffenden Operationen bereits Datensätze erzeugt haben.

Name	Operation	Scope	CPP-Kondition
...			
<i>privShowError</i>	Funkdef	0	1
<i>uiMessage</i>	Vardekl	<i>privShowError</i>	1

Tabelle 4.1: Neue Einträge der erweiterten Symboltabelle 1

Der in Abbildung 4.9 (S. 62) enthaltene Quellcode und der nebenstehende AST dienen als Beispiel für die Definition einer Union sowie der gleichzeitigen Erzeugung einer Typdefinition. Die Elemente Struktur und Union unterscheiden sich ausschließlich in der Anordnung der als Kombination zu interpretierenden Speicherbereiche und werden von *TypeChef* identisch verarbeitet. Der AST wird genutzt, um die Komplexität und Kombinierbarkeit dieses benutzerdefinierten Typs mit anderen Mechanismen aufzuzeigen. Die Zusammensetzung des abgeleiteten Typs geschieht in den Zeilen 4 bis 20. Eine innere Komponente wird durch den Pseudoknoten *NStructDeclarator* angezeigt. In *TypeChef* werden die Bezeichner *StackOS* (Z. 10) und *pxSave* (Z. 19) ermittelt und als nutzbare Komponenten der Symboltabelle hinzugefügt. Die Einflussnahme des Baumelementes *NTypedefName* (Z. 6) wird bereits im Rahmen des ersten Beispiels beschrieben.

Bei der Festlegung der Bestandteile von Struktur und Union können komplexe Zusammensetzungen angegeben werden. Folglich können benutzerdefinierte Typen wiederum Strukturen und Unions nutzen, aber auch definieren, wodurch die spezifische Verarbeitung geschachtelt zu wiederholen ist. Der Beispielquellcode nutzt in Zeilen 15 bis 19 die Struktur *TtaskSave*. Hierbei wird eine Referenzbeziehung zur Definition dieser Struktur hergestellt.

Die intern aufgebaute Symboltabelle enthält grundlegend alle Operationen, die ein Element der Programmiersprache C mittels einer spezifischen Operation erschaffen. Im Rahmen des in Abbildung 4.9 (S. 62) dargestellten Beispiels werden durch *TypeChef* drei neue Einträge erzeugt (vgl. Tabelle 4.2, S. 62). Für *StackOS* und *pxSave* wird je ein Datensatz durch das Bereitstellen einer Struktur-/ Union-Komponente generiert. Bei der Beschreibung der Union wird gleichzeitig eine globale Typdefinition *TosData* ermittelt und als weiterer Punkt eingetragen. Auch für dieses zweite Beispiel werden nur die neu erzeugten Einträge angezeigt.

1	<code>typedef union</code>	1	<code>: 166 : NTypeDef</code>
2	<code>{ Tchar StackOS[20];</code>	2	<code>: 102 : NDeclaration</code>
3	<code>struct TtaskSave pxSave;</code>	3	<code>typedef : 4 : "typedef"</code>
4	<code>} TosData;</code>	4	<code>union : 11 : "union"</code>
		5	<code>0 : 7 : LCURLY</code>
		6	<code>: 98 : NTypeDefName</code>
		7	<code>Tchar : 27 : ID</code>
		8	<code>: 101 : NStructDeclarator</code>
		9	<code>: 100 : NDeclarator</code>
		10	<code>StackOS : 27 : ID</code>
		11	<code>[: 34 : LBRACKET</code>
		12	<code>: 165 : NExpr</code>
		13	<code>20 : 140 : Number</code>
		14	<code>] : 35 : RBRACKET</code>
		15	<code>struct : 10 : "struct"</code>
		16	<code>TtaskSave : 27 : ID</code>
		17	<code>: 101 : NStructDeclarator</code>
		18	<code>: 100 : NDeclarator</code>
		19	<code>pxSave : 27 : ID</code>
		20	<code>} : 8 : RCURLY</code>
		21	<code>: 99 : NInitDecl</code>
		22	<code>: 100 : NDeclarator</code>
		23	<code>TosData : 27 : ID</code>
		24	<code>; : 9 : SEMI</code>

Abbildung 4.9: Quellcodeauszug und AST Beispiel 2 (Femto OS)

Name	Operation	Scope	CPP-Kondition
...			
StackOS	S/U-Komponente	0	1
pxSave	S/U-Komponente	0	1
TosData	Typedef	0	1

Tabelle 4.2: Neue Einträge der erweiterten Symboltabelle 2

Die in Abbildung 4.9 dargestellte Definition zeigt einen von verschiedenen Fällen auf, inwiefern die Kombination mit anderen Mechanismen von C das Ergebnis der Referenzanalyse beeinflusst. Die folgenden Möglichkeiten erläutern die Optionalität des Bezeichners und die Auswirkungen der kombinierten Definition/ Deklaration:

Bezeichner: Der AST in Abbildung 4.9 beschreibt eine unbenannte Union, da nach dem Literal *union* direkt die Komponentenbeschreibung vollzogen wird. Hierbei ist es jedoch möglich, einen Bezeichner einzuführen, um den Typ im weiteren Verlauf nutzbar zu machen.

Gleichzeitige Operationen: Im Zuge der Definition können gleichzeitig elementerschaffende Operationen realisiert werden. Im Beispiel wird die Union unmittelbar durch den vorangestellten Knoten *NTypeDef* zum Erstellen einer Typdefinition verwendet. Die Grammatik kapselt den synonymen Bezeichner *TosData* in den Teilbaum unter *NInitDecl*. Sobald die Beschreibung der Union nicht mit der Typdefinition kombiniert ist, fungiert dieser Teilbaum als simultane Deklaration einer Variable des hierbei erschaffenen Typs.

Je nach Optionalität, Kontext und Kopplung mit anderen Mechanismen muss die Referenzanalyse in *TypeChef* spezifische Routinen anstoßen. In C können weiterhin Aufzählungstypen erschaffen werden. Eine detaillierte Beschreibung wird an dieser Stelle nicht durchgeführt, da sich die Analyse ausschließlich für den Pseudoknoten („enum“) und die innere Struktur ändert. Die Zusammensetzung besteht aus einer Abfolge von Bezeichnern, die in der Art der Verwendung als Variablen angesehen werden können.

4.3.5 Bildung von Referenzpaaren

Im Rahmen der Referenzanalyse werden die elementspezifischen Operationen, die dessen Erschaffung bewirken, in die Symboltabelle eingefügt. Parallel zur Sammlung von existierenden Elementen wird im Verlauf der AST-Traversierung gleichzeitig jede Nutzung, wie beispielsweise ein Funktionsaufruf oder der Einsatz einer Struktur bei dem Anlegen einer Variable, verfolgt. Hierbei werden charakteristische Muster im AST als Indikator für eine gesonderte Verarbeitung verwendet. Das generelle Vorgehen wird im vorangegangenen Abschnitt anhand von Beispielen skizziert.

Die Referenzanalyse stützt sich auf die extrahierten Informationen und beginnt die Suche nach einem passenden Gegenstück in der Symboltabelle. Die Beziehung des Zugriffes und der Erstellung eines Elementes stützt sich auf den Bezeichner als eindeutiges Merkmal. Zusätzlich ist der jeweilige Gültigkeitsbereich zu beachten, wodurch die Referenzbeziehung ausschließlich zu an dieser Stelle sichtbaren Elementen hergestellt wird. In Kapitel 3.4.2 werden die bestehenden Verbindungen für die durch *TypeChef* untersuchten Quellcodeelemente beleuchtet. Die verschiedenen Paarungen werden in Tabelle 4.3 zusammengefasst.

Elementerstellung	Elementzugriff
Funktionsdeklaration/-definition	Funktionsaufruf
Definition eines benutzerdefinierten Typs (Struktur, Union und Enum)	Verwendung des Typs
Variablendeklaration (elementarer und benutzerdefinierter Typ)	Variablenzugriff
Definition von Struktur-/ Union-Komponenten	Zugriff auf die internen Komponenten
Definition von Aufzählungstyp-Konstanten	Zugriff auf die Mitglieder der Aufzählung
Erstellen von Typdefinitionen	Nutzung des eigenen Typs

Tabelle 4.3: Beziehungen zwischen den Erstellungs- und Zugriffsoperationen eines Elementes

Die angestoßene Suche in der Symboltabelle kann als erste Stufe der Typfehlerprüfung angesehen werden. Eine ergebnislose Suche lässt auf einen ziellosen Zugriff im Quellcode schließen, der je nach vorhandener Annotation in einer gewissen Anzahl

an Konfigurationen eine Inkorrektheit auslöst. Das Auftreten einer solchen Situation wird durch *TypeChef* registriert und am Ende der Bearbeitung einer Datei in Form eine Warnung ausgegeben, sodass die entsprechenden Elemente zurückverfolgt werden können. An dieser Stelle sind andere Problemquellen, die während der Phase des partiellen C-Präprozessors oder der Informationssammlung im AST hervorgerufen werden, nicht auszuschließen. Diesbezüglich wird die Problematik in Abschnitt 6.1 nochmals aufgegriffen und die Herkunft der Fehler diskutiert.

Grenzen der Paarbildung

Bei der Zusammenstellung der Referenzpaare wird eine Begrenzung in *TypeChef* erreicht. Sobald für eine Elementnutzung mehrere Erzeugungsoperationen in der Symboltabelle enthalten sind, ist die im Paar getroffene Zuordnung nicht mehr eindeutig. Eine elementbezogene Ausgabe dokumentiert eine Mehrfachzugehörigkeit und fordert den Benutzer zur Überwachung der jeweiligen Untersuchung auf. Ein Zugriff steht somit einer Liste von alternativ infrage kommenden Definitionen/ Deklarationen gegenüber. Die Referenzanalyse solcher Alternativen wird durch den Funktionsumfang von *TypeChef* nicht vollständig unterstützt. In einem gesonderten Vorgang werden die Bestandteile der Menge einzeln als Referenzpaar mit dem Zugriff ausgewertet. Die Fehlerfreiheit wird herausgestellt, wenn mindestens eines der Paare keinen Typfehler verursacht. Die mit dieser Begrenzung verbundenen Probleme und die hierbei notwendigen Modifikationen werden in Kapitel 6.3 näher erläutert.

Die Programmiersprache C erlaubt mit Hilfe von Marken (engl. Label) das Überspringen von Verarbeitungsschritten. Ein Label wird durch *TypeChef* als eine Variable erkannt, die jedoch erst im weiteren Verlauf des aktuellen Gültigkeitsblockes erschaffen wird. Für eine Elementnutzung wird die Paarbildung mit den Informationen der aktuellen Symboltabelle vollzogen. Folglich wird der Sprung zu einer Marke als Element ohne Zuordnung in die Liste der Fehler aufgenommen. Diese Problematik wird im Rahmen der Fallstudie sowie in Kapitel 6 aufgegriffen und mögliche Lösungen dargestellt.

Ist *TypeChef* in der Lage eine korrekte Verbindung herzustellen, wird für das Referenzpaar die CSP-Transformation eingeleitet und die Resultate auf das Auftreten eines Typfehlers überprüft.

4.4 CSP-Transformation der Annotationen

Für ein fokussiertes Quellcodeelement sind die an der Annotation beteiligten CPP-Bedingungen zu untersuchen und auf das Vorhandensein eines Typfehlers zu testen. Die Hauptaufgabe für diesen Arbeitsschritt besteht in der Überführung der Direktiven in ein Format, das für den CSP-Solver verständlich ist. Darauf aufbauend kann eine Aussage über das Auftreten eines Typfehlers für die in Beziehung stehenden Makros getroffen werden.

Die Wahl des zu verwendeten CSP-Solvers beeinflusst das Format sowie das Vorgehen für die Formulierung der Bedingungen. Im Rahmen dieser Arbeit wird die Biblio-

thek *Choco*⁶⁷ genutzt, die eine Java-basierte Definition, Analyse und Berechnung von Bedingungserfüllungsproblemen ermöglicht [JPC⁺08].

Der Choco-Solver benötigt ein vollständiges Modell, ein *CPModell*, welches das zu überprüfende Bedingungserfüllungsproblem umfasst. Das Modell setzt sich aus den nachstehenden Bestandteilen zusammen:

Variablen: Die Menge der beteiligten Variablen, die bei der Problembeschreibung verwendet werden.

Variablendomänen: Die Domäne einer Variable legt bei deren Definition den zulässigen Wertebereich fest.

Bedingungen: Die Menge der Bedingungen beschreibt das Problem unter Verwendung der durch die jeweilige Domäne begrenzten Variablen.

Die Annotation eines Quellcodeelementes erfolgt durch die Anweisungen der bedingten Übersetzung. Die darin enthaltenen Bedingungen zeigen die Abhängigkeit von den verwendeten Makros an. Für die Untersuchung der Korrektheit muss jeder Ausdruck, der die Präsenz beeinflusst, in das von *Choco* vorgegebene Format überführt werden.

4.4.1 Überführung eines Direktivenausdruckes

Aus einem Ausdruck, der in einer Direktive hinterlegt ist, müssen die relevanten Bestandteile für des CSP extrahiert werden. Der Parser erschafft durch die Funktionserweiterung einen AST, der die Zusammensetzung des Ausdruckes widerspiegelt. Der AST wird wiederum durch einen Visitor verarbeitet, der für einen besuchten Knoten in einer Fallunterscheidung über den Tokentyp die notwendige Weiterverarbeitung anstößt. In diesem Schritt werden die Operanden im Rahmen des jeweiligen Typs betrachtet:

Makro als Ganzzahl: Ein Makro wird durch das Token *ID* gekennzeichnet. Das Auftreten des Typs bewirkt die Definition einer Choco-spezifischen *IntegerVariable*, die den Makrobezeichner als Namen erhält.

Makrodefinition: Die Direktiven, welche die Definition eines Makros abfragen, werden im AST zu *PPdefined* oder *PPnotdefined* vereinheitlicht. Die Pseudobezeichnungen lösen während der Traversierung das Erschaffen einer Variablen für das betrachtete Makro aus. Je nach Überprüfung der An- oder Abwesenheit des Makros wird durch die Constraint *neq* auf Ungleichheit zu 0 beziehungsweise durch die Choco-Constraint *eq* auf die Gleichheit zu 0 getestet.

Konstante: Der GNU-C-Lexer ordnet einer auftretenden Konstante ein kennzeichnendes Token zu. Sobald der Visitor einen konstanten Wert im Bereich Integer ermittelt, wird für den Solver eine *constantVariable* generiert und der jeweilige Wert hinterlegt.

⁶ <http://www.emn.fr/z-info/choco-solver/index.html>

⁷ Dokumentation ist verfügbar unter: <http://www.emn.fr/z-info/choco-solver/tex/choco.pdf>

Die auf diesem Weg erstellten Variablen und Beziehungen werden mit Hilfe eines Stacks verwaltet und dienen als Quelle für das Anlegen der Choco-Elemente *Constraint* und *IntegerExpression*, die im Solver je nach Operator zur Verknüpfung von Variablen untereinander aber auch zur Verknüpfung von Bedingungen genutzt werden. Die bereits vorhandene Fallunterscheidung wird um zusätzliche Möglichkeiten für die Operatoren erweitert:

Arithmetische Verknüpfung: Die arithmetischen Operatoren werden für Choco durch die Definition einer *IntegerExpression* erfasst. Hierbei können unterschiedliche Kombinationen von Variablen und *IntegerExpression* in Beziehung gesetzt und die dazugehörigen arithmetischen Gleichungen bearbeitet werden.

Relationale Verknüpfung: Die im CPP erlaubten Verknüpfungen zum Beschreiben von Relationen können direkt in existierende Constraints überführt werden. Die Vergleichsoperatoren akzeptieren wiederum die Operanden: *Variable* und *IntegerExpression*, um diese in Beziehung zueinander zu setzen.

Logische Verknüpfung: Im C-Präprozessor werden die logischen und-/ oder-Verknüpfungen für die kombinierte Auswertung von mehreren Bedingungen eingesetzt. Im Choco-Solver werden hierfür die Constraints *and/ or* bereitgestellt, worin die bereits ermittelten Constraints konjunktiv beziehungsweise disjunktiv verbunden werden.

Alle direkt umsetzbaren Verknüpfungen nutzen den Stack als Bezugsquelle für die zu verbindenden Elemente. Eine *Variable* in ihrer möglichen Ausprägung wird in das Format des Solvers übertragen und dem Stack hinzugefügt. Gelangt die Verarbeitung an eine Verknüpfung, so werden die zwei letzten Elemente des Stacks in der Funktion eines Operanden in die jeweilige Beziehung gesetzt. Folglich wird ein neues Element, hier *IntegerExpression* oder *Constraint*, generiert und in den Stack als oberstes Element eingetragen.

Die Ausdrücke in den Direktiven erlauben die Klammerung von Gleichungen, um die enthaltenen Berechnungen aus der gewöhnlichen Operatorenhierarchie herauszulösen und diesbezüglich die Auswertungsreihenfolge zu manipulieren. Eine solche Situation wird durch den Pseudoknoten *NExpressionGroup* als weiterer Fall für die Verzweigung eingeführt, wodurch eine öffnende Klammer in der Grammatik symbolisiert wird. Jedes Auftreten des Knotens bewirkt das Erschaffen einer weiteren Visitor-Instanz, um die geklammerte Gleichung in die benötigten Modellbestandteile zu überführen. Zwischen den Instanzen existiert ein Informationsfluss, der am Ende der Bearbeitung einer inneren Teilgleichung die Rückführung der ermittelten Variablen und Constraints sicherstellt.

Auf dem Stack ist nach dem Abschluss der AST-Traversierung eine einzelne Choco-Constraint enthalten, die den Ausdruck der Direktive in dem für den Solver verständliches Format formuliert. Alle in der Bedingung verwendeten Variablen werden zusätzlich in einer separaten Liste zusammengeführt. Die Liste gewährleistet, dass eine *Variable* einzigartig ist und nicht mehrfach definiert wird. Folglich wird beim Erstellen einer *Variable* auf deren Vorkommen in der Liste geprüft. Die Vorgehensweise verhindert eine mögliche Mehrfachdefinition und das daraus resultierende Fehlverhalten

des Lösungssystems, wenn eine Variable durch mehrere, gleichzeitige Instanzen unterschiedliche Werte in einem Solver-Durchlauf annimmt und irrtümlich einen Typfehler voraussagt.

Grenzen des CSP-Transformationssystems

Der beschriebene Mechanismus in *TypeChef* ist auf die Art von Direktiven beschränkt, die direkt in Bedingungen oder Ausdrücke des Choco-Solvers abgebildet werden können. Die Menge der bitweisen Operatoren wird nicht unterstützt und müssten Java-seitig realisiert werden. Nach der Auflösung der Operation kann das Ergebnis als normale Variable in die bisherige Verarbeitung übergeben werden. Weiterhin erlaubt der CPP die Nutzung von Charakterkonstanten als Operanden einer Gleichung. Hierbei muss das Transformationssystem eine Umwandlung vornehmen, die den Buchstaben als eine Ganzzahl interpretiert und das Ergebnis, wie bereits beschrieben, in der Zusammensetzung von Bedingungen zu verwenden.

Die Überführung eines Ausdruckes ist ein mehrfach zu wiederholender Vorgang, um im nächsten Schritt, die Hierarchie von CPP-Direktiven in das CSP-Modell zu integrieren.

4.4.2 Konstruktion der Hierarchiebedingungen

Im Verlauf der Informationsaufbereitung wird jedem Element eine Eigenschaft zugeordnet, die die Abhängigkeit von der CPP-Hierarchie und somit die notwendigen Gegebenheiten für den Verbleib im Quellcode ausdrückt. In Kapitel 3.5 wird das Prinzip des Zusammenspiels der CPP-Anweisungen erläutert, um den Hierarchiepfad zum betrachteten Element in eine Aussage zu transferieren und die Gesamtbedingung für das Erreichen des Quellcodebereiches zu erschaffen.

Jede Ebene wird im Analysewerkzeug durch eine Liste beschrieben, die alle an der Auswertung beteiligten Direktiven, speziell den AST des darin formulierten Ausdruckes, enthält. Aus dem AST wird, nach dem in Abschnitt 4.4.1 dargelegten Prinzip, eine Menge von CSP-Variablen erzeugt, die durch eine CSP-Bedingung in Beziehung zueinander gesetzt werden. Über die Anzahl der Listenelemente wird für jede beteiligte Direktive entschieden, inwiefern diese in normaler oder in negierter Form zu behandeln ist. Umfasst die Liste nur ein Element, so ist die Direktive vom Typ `#ifdef`, `#ifndef` oder `#if` und symbolisiert den Beginn der aktuellen Ebene. Bei einer Anzahl größer 1 wird der Ausdruck des letzten Listenelementes mit der Negation der Ausdrücke 1 bis $n-1$ zu einer Ebenenaussage zusammengesetzt. Die aus der Überführung resultierende CSP-Bedingung wird durch die Choco-Constraint *not* negiert. Zusätzlich wird für die Liste ein Spezialfall vermerkt, der eintritt, wenn in den `#else`-Fall der Ebene verzweigt wird, da dann alle gleichrangigen Direktiven mittels der Constraint *not* negiert und in die Aussage aufgenommen werden.

Nach der Verarbeitung der Gegebenheiten einer Ebene und der Überführung der verwendeten Direktiven in das Choco-Format ist eine Ebenenbedingung zu konstruieren. Hierbei werden alle transformierten Ausdrücke zusammengeführt (vgl. Kapitel 3.5.1). Der Solver stellt für die Durchführung der konjunktiven Verknüpfung die Constraint *and* bereit und erschafft somit eine ebenenspezifische Gesamtbedingung.

Ein zu untersuchendes Quellcodeelement kann durch mehrere, verschachtelte C-Präprozessorebenen annotiert werden. Alle beteiligten Ebenen, die während der Informationsaufbereitung ermittelt werden, müssen in das CSP-Format übersetzt werden. Das erläuterte Vorgehen wird für jede Ebene wiederholt und eine ebenenspezifische Choco-Constraint erstellt. Die Resultate werden wiederum unter Verwendung der Constraint *and* verbunden, um die vollständige Hierarchie in eine Gesamtbedingung abzubilden.

Das mehrstufige Verfahren ist die Umsetzung der in Kapitel 3.5.1 definierten Präsenzfunktion. Das Analysewerkzeug erzeugt als Zwischenprodukt ein System aus Variablen, die die zur Konfiguration verwendeten Makros verkörpern sowie eine Choco-Constraint, welche die Gesamtbedingung für das Erreichen des betrachteten Quellcodeelementes festlegt und vom Choco-Solver für die Prüfung auf einen Typfehler benötigt wird.

4.5 Formulierung des Bedingungserfüllungsproblems

Ein zusammengestelltes Paar wird in eine separate Abarbeitung übergeben, um eine Aussage über die Korrektheit des dahinterliegenden Quellcodes in allen Varianten des Projektes zu treffen. Die Bestandteile des Paares werden hinsichtlich ihrer Annotation untersucht. Diesbezüglich wird die CPP-Hierarchie beider Elemente in eine Gesamtbedingung des CSP-Solvers Choco transformiert (vgl. Kapitel 4.4.1). Für die Bestandteile eines Paares wird je eine Choco-Constraint generiert, welche die Präsenz des Elementes sowie eine Menge von Variablen und Konstanten, mit deren Hilfe die Konfiguration durchgeführt wird, festlegt.

Die Analyse eines Typfehlers definiert ein Bedingungserfüllungsproblem. In Kapitel 3.5.3 wird ein solcher Fehler mit Hilfe der Präsenzfunktion eines Elementes symbolisiert. Die beschriebene Situation muss in der Form einer Choco-Constraint vorliegen, um durch den Solver auf Lösbarkeit geprüft zu werden.

Aus der Aussage über einen Typfehler:

$$FM \wedge \neg(\text{Präsenz}(\text{Kondition}(E_{\text{Zugriff}})) \Rightarrow \text{Präsenz}(\text{Kondition}(E_{\text{Erstellen}}))) \Rightarrow \text{Typfehler}$$

lässt sich die Gesamtgleichung durch die direkte Umformung in das CSP-Format realisieren. Die Präsenzbedingung für Zugriff und Erstellen sind die Bestandteile eines Paares, welche bereits aufbereitet vorliegen (vgl. Kapitel 4.4.1). Im Funktionsumfang des Solvers wird mit der Constraint *implies* eine Möglichkeit bereitgestellt, den inneren Gleichungsteil, insbesondere die Implikation, direkt umzusetzen. Als Argumente werden die bereits transformierten CPP-Hierarchien für die zu betrachtenden Elemente übergeben. Letztendlich ist die Constraint *not* auf die Implikation anzuwenden, um den Widerspruchsbeweis zu beschreiben. Das Merkmalmodell (FM) wird als weiterer Bestandteil konjunktiv in die Gleichung integriert. Falls ein Modell für eine zu untersuchende Anwendung vorhanden ist, muss eine dem Choco-Format entsprechende Constraint formuliert werden. Die modellbeschreibende Constraint wird durch eine Choco-Konstante mit dem Wert 1 initialisiert, sodass ein fehlendes Modell den Ablauf der Prüfungen von *TypeChef* nicht beeinflusst.

Das Resultat beinhaltet die Formulierung eines Typfehlers. Die Menge aller Konfigurationsvariablen und Konstanten wird in das CPMoell eingepflegt. Des Weiteren ist die zusammengesetzte Gesamtbedingung, in der alle beteiligten Zwischenkonditionen enthalten sind, in das Modell eingelesen. Der Choco-Solver nutzt die Problembeschreibung und beginnt mit der automatisiert ablaufenden Lösungsbestimmung. Sobald eine Variablenbelegung herausgestellt werden kann, die zur Erfüllung der Typfehlergleichung führt, ist die Annotation des Referenzpaares nicht korrekt und erzeugt in mindestens einer konstruierbaren Variante einen Fehler. Das entwickelte Analysewerkzeug generiert im Fehlerfall eine Ausgabe über die in Konflikt stehenden Mitglieder des Referenzpaares. Das aktuell betrachtete Paar ist korrekt annotiert und in allen Konfigurationen fehlerfrei, wenn keine Belegung ermittelt werden kann.

4.6 Vollständige Analyse eines Projektes

Die im Einzelnen in einem Projekt vorhandenen Dateien werden der vollständigen Analyse unterzogen. Jedes im Projektverzeichnis enthaltene Dokument der Programmiersprache C wird an den Arbeitsschritt des partiellen C-Präprozessors übergeben. Dementsprechend werden alle Anweisungen der bedingten Übersetzung auskommentiert, die Abarbeitung der verbleibenden Direktiven mit Hilfe des GCC durchgeführt und nachfolgend alle mit Kommentaren versehenen Direktiven wieder in den Quellcode reintegriert.

In den vorverarbeiteten Dateien wird, im Zuge der Informationsaufbereitung, der AST auf die Verwendung von disziplinierten Annotationen überprüft und in dem Fall einer Verletzung durch Fehlerausgaben angezeigt. Der GNU-C-Parser sichert die Einhaltung der Disziplin und zeigt eine Verletzung durch den Abbruch der syntaktischen Analyse an. Die Einhaltung der Annotationsebenen wird zusätzlich durch den Disziplin-Visitor abgesichert.

Insofern die Markierungen in einer Datei gänzlich der Disziplin entsprechen, werden alle Referenzpaare im Kontext des CPP, speziell die makroabhängigen Gegebenheiten unter denen die Elemente ein Teil des Dokumentes sind, herausgestellt. Jede ermittelte Kombination aus Erstellung und Zugriff ist auf die Verursachung eines Typfehlers zu untersuchen.

Die Typfehlerprüfung ist für alle auftretenden Referenzpaare zu wiederholen. Für jedes Paarelement wird die an der Präsenz beteiligte Direktivenhierarchie überführt, die Aussage für eine fehlerbehaftete Referenzbeziehung formuliert und in das Lösungssystem eingelesen. Für das hierbei erschaffene Bedingungserfüllungsproblem werden alle Belegungen der genutzten Variablen bewertet, um im Fehlerfall eine Lösung zu erhalten. Das Analysewerkzeug dokumentiert den Verlauf der Abarbeitung mit Ausgaben, die einen Rückschluss auf die am jeweiligen Test beteiligten Elemente zulassen. Ein in der untersuchten Anwendung vorhandener Fehler kann somit gefunden werden, wodurch die Möglichkeit besteht, diesen zu entfernen und alle generierbaren Varianten des Projektes hinsichtlich des fokussierten Referenzpaares zu korrigieren.

4.7 Kapitelzusammenfassung

Der Inhalt des vorliegenden Kapitels dient der Beschreibung von Implementierungsdetails des im Rahmen dieser Arbeit entwickelten Werkzeuges *TypeChef*. Speziell wird das konkrete Vorgehen im partiellen C-Präprozessor aufgezeigt.

Anschließend wird der Parsergenerator ANTLR genutzt, um ein Quellcodeanalyse-system für variabel gestaltete C-Dokumente zu erschaffen. Der erweiterte Parser generiert einen AST, der zur Bewältigung der Disziplinsicherstellung und insbesondere für die Extraktion von Referenzen benötigt wird.

Anhand von spezifischen Mustern im AST werden die zu untersuchenden Elemente herausgestellt und hinsichtlich der existierenden Beziehungen zu Referenzpaaren kombiniert.

Für die verbleibenden Phasen von *TypeChef* stehen die Annotationen der Paare im Fokus. Die zur Markierung genutzten Bedingungen definieren auf Grund der anwendbaren Komplexität ein CSP. Die statische Referenzanalyse prüft alle konstruierbaren Varianten mit Hilfe eines CSP-Solvers. Hierbei werden die Makros und Bedingungen, je nach Zusammensetzung, in das Format des verwendeten Solvers Choco transformiert, wodurch alle CPP-Direktiven, die den Verbleib beschreiben, automatisiert geprüft werden können.

Durch die Formulierung der Aussage über einen Typfehler wird ein Modell festgelegt, dass das Lösungssystem auf eine erfüllende Variablenbelegung untersucht. Solange der Choco-Solver keine Lösung ermittelt, wird das jeweilige Paar als fehlerfrei eingestuft.

KAPITEL 5

Evaluation

In diesem Kapitel wird im Rahmen einer Fallstudie die Funktionsweise von *TypeChef* überprüft. Zur Veranschaulichung von quellcodebezogenen Sachverhalten wurde in dieser Arbeit bisher die Anwendung Femto OS verwendet. Für die durchgeführten Fallstudien ist dieses Projekt nicht geeignet. Femto OS beinhaltet eine feingranuläre Nutzung des CPP. Eine Vielzahl der Annotationen wird nach den Festlegungen in *TypeChef* als undiszipliniert eingestuft. Ohne die Integration einer automatisierten Disziplinierung müssen die Änderungen an den Quellcodedokumenten manuell vorgenommen werden. Auf Grund des resultierenden Aufwandes ist Femto OS nicht als Fallstudie geeignet.

TypeChef wurde zur Analyse von Referenzen in bestehenden Anwendungen entwickelt. Das Ziel dieses Kapitels ist Durchführung der statischen Referenzanalyse für die existierenden Projekte *boa* und *gretl*. Auf der Basis dieser Anwendungen wird herausgestellt, inwiefern Typfehler durch *TypeChef* ermittelt werden können. Die Resultate der Untersuchung werden für ausgewählte Fehlersituationen manuell überprüft und die jeweiligen Konfigurationen konstruiert. Mit Hilfe der Analyse von *boa* wird die Anwendbarkeit von *TypeChef* auf den Quellcode von reellen Systemen getestet, um Probleme und Grenzen der derzeitigen Verarbeitung ausfindig zu machen.

5.1 Statische Referenzanalyse in *boa*

Die Anwendung *boa*¹(Version 0.94.13) wird auf das Vorhandensein von fehlerhaften Präprozessorannotationen bei Quellcodereferenzen untersucht. Der HTTP-Webserver ist frei verfügbar und quelloffen. Der dazugehörige Quellcode verteilt sich über insgesamt 38 Dateien, wobei 27 dieser Dokumente (*.c) die direkte Implementierung der Elemente beinhalten und die verbleibenden 11 Header-Dateien zur Deklaration der Elemente dienen. Durch die Direktiven wird die Anwendung hinsichtlich des enthaltenen Funktionsumfanges angepasst. Da für *boa* kein Merkmalmodell vorhanden ist, werden die bestehenden Annotationen manuell hinsichtlich ihrer Funktionalität und erkennbaren Abhängigkeiten analysiert. Der Quellcode der Fallstudie wird durch den CPP verändert, um während des Programmablaufes verschiedene Debug-Mechanismen sowie Logging-Ausgaben zu nutzen. Des Weiteren wird die Wahl eines Hash-Verfahrens getroffen und die Verwendung des erweiterten Internet-Protokolls IPv6 festgelegt. Aus den verbleibenden Makros können keine Eigenschaften abgeleitet werden. Die Suche

¹ <http://www.boa.org/>

nach Abhängigkeit zwischen den einzelnen Makros blieb ergebnislos, sodass auf keine gegenseitige Beeinflussung der Merkmale geschlossen werden kann.

Auf der Grundlage einer Suche nach den charakteristischen Mustern können Aussagen über die Anzahl und Verteilung der Präprozessoranweisungen über alle Dokumente des Projektes getroffen werden. In 25 der insgesamt 38 Dateien werden die CPP-Direktiven zur Erzeugung von Variabilität eingesetzt. Hierbei werden 97 Vorkommen der öffnenden Anweisungen `#if`, `#ifdef` und `#ifndef` ermittelt. Ausgehend von dieser Art von Direktiven wird in 26 Fällen direkt in einen alternativen Konfigurationsblock verzweigt. Weiterhin sind 97 Anweisungen des Typs `#endif` enthalten, um das Ende eines variablen Abschnittes zu kennzeichnen. Jede dieser Anweisungen wird verarbeitet und in einen Kommentar verschoben.

5.2 Partieller C-Präprozessor

Die Anpassung eines Projektes ist nicht auf die Auswahl von funktionellen Merkmalen beschränkt. Die Entwicklung einer Anwendung kann für verschiedene Plattformen oder Betriebssysteme realisiert werden. Daraus resultiert eine größere Anzahl an Makros, um die jeweils spezifische Funktionalität, wie das Speichern von Daten in einer Datei, zu kapseln. Diesbezüglich sind zusätzlich systemabhängige Header-Dateien erforderlich, um den Zugriff zu bewerkstelligen. Da die benötigten Standardinklusionen nicht im GCC enthalten sind, können diese Abhängigkeiten nicht aufgelöst werden. Die Phase des partiellen Präprozessors ist hierfür mit einer Liste von Makros versehen, die im Vorgang des Auskommentierens von Direktiven der bedingten Übersetzung nicht verarbeitet werden. Der GCC wertet die Makrobedingungen hinsichtlich des vorhandenen Systems aus und verhindert die fehlerhafte Ausführung der Inklusionen. Das beschriebene Vorgehen ermöglicht die Durchführung der anschließenden Analyse. Der Mechanismus ist auf einfache Direktivenblöcke begrenzt. Folglich müssen komplexere Strukturen manuell überprüft und angepasst werden. Die plattformabhängigen CPP-Anweisungen werden in Kapitel 6.2 ausführlicher betrachtet.

5.3 Disziplin der Annotationen

Die partiell vorverarbeitete Datei beinhaltet nun ausschließlich den C-Quellcode, der mit bedingten Anweisungen durchsetzt ist. Ein Auftreten solcher Anweisungen ist generell an jeder Position im Quellcode möglich. Für das zu analysierende Projekt ist die Einhaltung der Disziplin gefordert. Jede Verletzung ist durch eine manuelle Veränderung der Annotation zu restrukturieren, um der Disziplin zu entsprechen. Der Vorgang wird in verschiedenen Schritten der Projektanalyse durch die dort verfügbaren Informationen gestützt.

Anzahl der Direktiven

Im Rahmen der Voruntersuchung wird durch das Ergebnis einer Mustersuche die Anzahl der einen Variabilitätsblock öffnenden und der diesen abschließenden Anweisung gegenübergestellt. Die Gesamtmenge der beiden Direktivenarten besteht jeweils aus

97 Elementen, wonach jeder enthaltene CPP-Bereich einen Abschluss erhält. Durch eine detaillierte Suche in jeder einzelnen Datei ist sicherzustellen, dass eine Ausgeglichenheit im Hinblick auf das Direktivenverhältnis besteht. Für boa wird in diesem Zusammenhang keine fehlerhafte konstruierte Annotation herausgestellt, da die Elemente der betrachteten Mengen ausgeglichen sind.

Prüfung durch den Parser

Die Erweiterung der bestehenden Grammatik ist auf die Durchsetzung der Disziplin ausgerichtet und führt zu einem Ausführungsabbruch seitens des generierten Parsers. Demzufolge wird in der syntaktischen Analyse keine Regel gefunden, in die im Fall einer undisziplinierten Annotation verzweigt werden kann.

Für das Untersuchungsobjekt werden während des Parservorganges in der Summe 8 Fehler gefunden. Die 8 undisziplinierten Markierungen verteilen sich über 6 Dateien. Eine Verletzung der Disziplin wird auf verschiedenen Ebenen der Granularität festgestellt, deren Transformation meist mittels der Replikation des an dieser Stelle betroffenen Quellcodes gelöst wird.

Das Quellcodebeispiel in Abbildung 5.1 beinhaltet eine Markierung, die einen Parserfehler verursacht. Ein if-/else-Block wird als zusammengehöriger Bereich unterhalb eines Statements abgehandelt, wodurch ausschließlich eine vollständige CPP-Umrahmung zulässig ist. Der in den Zeilen 3 bis 6 annotierte else-Block stellt somit eine Verletzung der Disziplin dar.

```

1  if (req->pathname[_dmalloc_strlen(req->pathname) - 1] == '/')
2      execl(dirmaker, dirmaker, req->pathname, req->request_uri, ((void *)0));
3  #ifndef GUNZIP
4      else
5          execl(GUNZIP, GUNZIP, "--stdout", "--decompress", req->pathname, ((void *)0));
6  #endif

```

Abbildung 5.1: Beispiel für eine die Disziplin verletzende Annotation (boa)

In Abbildung 5.2 ist das Ergebnis der Transformation dargestellt. Der Inhalt der Annotation wird um das Duplikat des entsprechenden if-Blockes erweitert (Z. 2-3). Weiterhin wird eine Alternativdirektive (Z. 6-9) erschaffen, die den Ursprungszustand für die Abwesenheit des Makros GUNZIP formuliert.

```

1  #ifndef GUNZIP
2      if (req->pathname[_dmalloc_strlen(req->pathname) - 1] == '/')
3          execl(dirmaker, dirmaker, req->pathname, req->request_uri, ((void *)0));
4      else
5          execl(GUNZIP, GUNZIP, "--stdout", "--decompress", req->pathname, ((void *)0));
6  #else
7      if (req->pathname[_dmalloc_strlen(req->pathname) - 1] == '/')
8          execl(dirmaker, dirmaker, req->pathname, req->request_uri, ((void *)0));
9  #endif

```

Abbildung 5.2: Ergebnis der manuellen Transformation

Neben variablen if-Bedingungen werden in boa auch die Signaturen von Funktionen je nach Konfiguration ausgetauscht. Hier ist der vollständige Funktionsinhalt zu duplizieren und den unterschiedlichen Köpfen zuzuordnen. In C besteht eine strikte Trennung zwischen der Liste der Deklarationen sowie der Statements. Zwei alternative Kombinationen dieser Listen verursachen Fehler bei der syntaktischen Analyse, da genau ein Exemplar erlaubt ist. Die Zusammensetzung der Annotationsblöcke ist dahingehend zu trennen, dass die Deklarationen sowie Statements jeweils einzeln in die Abhängigkeit zu CPP-Makros gestellt werden.

Insgesamt 8 manuelle Änderungen sind notwendig, um die Quellcodedokumente in eine parserverträgliche und disziplinierte Form zu überführen. Der erstellte AST wird im anschließenden Schritt auf die Korrektheit der verwendeten Markierungen hinsichtlich der Ebene, auf der sie verwendet werden, überprüft.

Disziplin-Visitor

Die Integrität der Annotationen wird durch den Vergleich der Anzahl von öffnenden und schließenden Direktiven sowie der Verarbeitung durch den angepassten Parser überprüft. Sobald der AST des zu untersuchenden Dokumentes lückenlos erstellt wird, vollzieht der Disziplin-Visitor eine Analyse der Konfigurationsabschnitte. Ein Abschnitt ist als diszipliniert anzusehen, wenn alle zuzuordnenden Direktiven in der identischen Tiefe im AST und somit auf gleicher Quellcodeebene auftreten. Als Resultat der Baumuntersuchung werden alle fehlerhaft ausgeführten Markierungen, speziell die verursachende CPP-Anweisung, dokumentiert.

Die Traversierung der Baumstrukturen, die für jede der partiell vorverarbeiteten Quellcodedateien in boa durchzuführen ist, ermöglicht es, eine Aussage über das Vorhandensein von unzureichend umschlossenen Codeabschnitten zu treffen. In einer der Dateien ist eine Markierung vorhanden, die durch den Parser akzeptiert, jedoch durch den Visitor als fehlerbehaftet aufgezeigt wird. Der Fehler entsteht durch den Einschub eines if-/else-Blockes, wodurch eine Liste von Anweisungen, die immer präsent sind, innerhalb des else-Falls dieses Einschubs abgehandelt werden. Mittels der Replikation wird die Zusammensetzung modifiziert und letztendlich durch den Disziplin-Visitor akzeptiert.

5.4 Ermittlung von Typfehlern

Als Ergebnis der Datenaufbereitung und Bildung der Paare werden alle notwendigen Informationen aus dem AST ermittelt und für die herstellbaren Beziehungen auf Fehler untersucht.

5.4.1 Funktion

Bevor eine Typfehlerprüfung für die Funktionen durchgeführt werden kann, ist die Verbindung zwischen einem der 2008 Funktionsaufrufe und einer, der 20342 dazugehörigen, Definitionen/ Deklarationen herzustellen.

Bereits in dieser Phase werden Fehler festgestellt. In 30 Fällen kann kein Element, in der Ausprägung einer Definition oder Deklaration, gefunden werden. Diese 30 Aufrufe führen ins Leere, sodass bereits erste Typfehler verursacht werden.

Für 1978 Aufrufe wird eine Untersuchung der vorhandenen CPP-Hierarchie durchgeführt. Ein Referenzpaar wird in ein CSP transformiert, wenn die beiden Elemente annotiert sind. In 234 Konstellationen ist die Transformation die Annotationen notwendig. Das Lösungssystem ermittelt für 204 der Paare einen Fehler. Alle inkorrekten Paare rufen im vollständigen Konfigurationsumfang einen Typfehler hervor, wobei der Funktionsaufruf trotz fehlender Definition/ Deklaration im resultierenden Quelldokument vorzufinden ist.

5.4.2 Variable

TypeChef ermittelt für die Anwendung *boa* insgesamt eine Anzahl von 9875 Variablen-deklarationen. Dem gegenüber stehen 7250 Zugriffe auf Variablen. In dieser Menge von Zugriffen sind weiterhin die Elementenutzungen von Aufzählungstyp-Konstanten sowie von Funktionszeigern enthalten, da keine Unterscheidung in der Verwendung getroffen werden kann.

Bei der Zusammenstellung der Referenzpaare werden für 52 Zugriffe keine dazugehörige Deklaration beziehungsweise Konstantwerte einer Aufzählung gefunden, die im Bezeichner sowie Gültigkeitsbereich übereinstimmen, obwohl die erweiterte Zuordnung den Einsatz eines als Variablenzugriff erkannten Funktionszeigers testet. Die manuelle Kontrolle der aufgezeigten Fehler bestätigt die Ungültigkeit der betroffenen Quellcodeabschnitte.

Die Überprüfung der Präprozessorabhängigkeit wird für 743 Paarungen als notwendig herausgestellt. Das Lösungssystem findet in 5 Fällen eine Variablenbelegung, die einen Typfehler hervorruft. Somit werden die CPP-Annotationen 738-mal korrekt angewendet.

5.4.3 Struktur-/ Union-Typ

In den partiell vorverarbeiteten Dokumenten werden 2553 Strukturen/ Union definiert. Weiterhin registriert das Analysewerkzeug 6179 Nutzungen dieser vom Benutzer definierbaren Konstrukte, um beispielsweise einen Variablentyp oder die Reinterpretation eines Speicherbereiches festzulegen.

Mehrere der untersuchten Dateien beinhalten 55 Nutzungen eines Struktur-/ Union-Typs, die nicht zugeordnet sowie mittels manueller Suche nicht ausfindig gemacht werden können, sodass nicht alle Quellcodevarianten fehlerfrei übersetzbar sind.

Für 792 Paarungen besteht das Potenzial für die Verursachung eines Typfehlers. In 184 Fällen wird ebendiese Situation durch *TypeChef* nachgewiesen und der bei der Definition von Struktur/ Union erstellte Typ wird in einem abweichenden Konfigurationskontext genutzt.

Struktur/ Union-Komponente

Innerhalb von Struktur/ Union werden 13984 Komponenten bereitgestellt, die im weiteren Verlauf der Abarbeitung 1300 Mal angesprochen werden.

Eine spezielle Komponente wird zwar verwendet, jedoch nicht definiert. Alle dadurch betroffenen Konfigurationen werden seitens des Compilers als invalide abgewiesen.

Der CSP-Solver führt für 164 Paare eine Erfüllbarkeitsprüfung aus. Hierbei ist die Heuristik bei der Zuordnung von Komponentenzugriffen zu beachten, durch die ein Paar nur gebildet wird, wenn eine eindeutige Zuordnung möglich ist. Der Zugriff auf 148 Komponenten verursacht einen Typfehler, da die umschließende Struktur/ Union vollständig beziehungsweise die innere Zusammensetzung in einem sich unterscheidenden CPP-Kontext existiert.

5.4.4 Aufzählungstyp

Während der Informationssammlung werden in der Summe 55 Definitionen eines Aufzählungstyps registriert. Diese Typen werden in keiner Weise nochmals genutzt, wodurch weitere Analysen dieser Elementart nicht durchzuführen sind.

Jede Nutzung einer Ganzzahlkonstante wird durch das Werkzeug als Variablenzugriff interpretiert und in der dortigen Abarbeitung behandelt.

5.4.5 Typdefinition

Im Untersuchungsobjekt boa werden abschließend die Typdefinition mit einer Anzahl von 4747 und der Einsatz dieser Synonyme in 21934 Fällen verglichen.

Die Zuordnung der Referenzpaare wird bereits durch den GNU-C-Parser abgesichert. Die intern mitgeführte Synonymtabelle verhindert den Zugriff auf nicht definierte Typdefinitionen. Dementsprechend werden alle Paare korrekt verknüpft.

Von den hergestellten Elementbeziehungen werden 1260 überführt und detailliert untersucht. Der Konditionenvergleich stellt für 481 Nutzungen einer Typdefinition heraus, dass ein Typfehler besteht und somit ein ungültiger Quellcode generiert werden kann.

5.4.6 Gesamtergebnis

Die Referenzanalyse für das Projekt boa hat neben den fokussierten Typfehlern auch nicht existente, jedoch referenzierte Elemente vorgefunden. In der Tabelle 5.1 (S. 77) werden die Resultate zusammengefasst. Innerhalb der 27 Quellcodedokumente werden die Daten für 36864 Paare gesammelt. Aus 3192 vom CSP-Solver untersuchten Konstellationen werden 1022 als inkorrekt und einen Typfehler hervorrufen eingestuft.

Im nächsten Abschnitt wird mit Hilfe von Quellcodeabschnitten, die durch *TypeChef* als nicht nachvollziehbar oder fehlerkonfiguriert gekennzeichnet werden, die Korrektheit der Untersuchungsergebnisse und somit die Funktionsweise des Werkzeuges bewertet.

Referenziertes Element	# RP	# NR	# CT	# TF
Funktion	1744	30	234	204
Variable*	6455	52	743	5
Struktur-/ Uniontyp	5332	55	792	184
Struktur-/ Unionkomponenten	1299	1	164	148
Aufzählungstyp	0	0	0	0
Typdefinition	21934	0	1260	481
Gesamt:	36864	138	3192	1022

RP - Referenzpaare, NR - keine Referenzzuordnung, CT - durchgeführte CSP-Tests, TF - Typfehler

* beinhaltet zusätzlich die Prüfung von Aufzählungstyp-Konstanten und Funktionszeigern

Tabelle 5.1: Zusammenfassung der Referenzanalyse für das Projekt boa

5.5 Detaillierte Fehlerbetrachtung

Die Untersuchung des Projektes boa durch *TypeChef* hat 138 nicht nachvollziehbare Elemente und 1022 inkorrekt konfigurierte Paare herausgestellt. Folglich werden insgesamt 1160 Typfehler ermittelt, wenn alle generierbaren Konfigurationen der Anwendung betrachtet werden. Im weiteren Verlauf des Abschnittes werden die Fehlergruppen genauer beleuchtet und die Analyseresultate anhand von Quellcodebeispielen diskutiert.

5.5.1 Typfehler durch fehlende Zuordnung

Die C-Dokumente beinhalten 138 Elemente, für die kein Gegenstück in der Symboltabelle enthalten ist. Generell wird diese Situation als ein Typfehler interpretiert.

Die Arbeitsweise von *TypeChef* bei der Zuordnung von Elementen ermöglicht keine Unterstützung von Sprungmarken, da keine zusätzliche Operation ausgeführt werden muss, die das Element vorher erschafft. In boa wird dieses Quellcodekonstrukt in 55 Fällen verwendet, wodurch die Anzahl der nicht nachvollziehbaren Elemente auf 83 reduziert wird. In Abschnitt 6.1 werden Ansätze zur Referenzanalyse von Marken vorgestellt.

Die verbleibende Menge sind somit Auslöser für fehlerhafte Konfigurationen. Die anschließend dargestellten Ausschnitte zeigen Beispiele für die Elemente mit einer fehlenden Zuordnung. Nach der Erläuterung von drei ausgewählten Fehlersituationen werden die Konfigurationen nachgestellt und auf die Existenz überprüft.

In der Abbildung 5.3 (S. 78) wird die Funktion *fprintf* (Z. 2) aufgerufen. *TypeChef* kann den letzten Parameter dieses Aufrufes nicht auflösen. Die manuelle Kontrolle bestätigt das Fehlen der Variable *h* im lokalen sowie globalen Gültigkeitsbereich. Im unteren Teil der Abbildung wird ein Zugriff auf die bereits in Abschnitt 5.4 als einzige nicht nachvollziehbare Komponente *__ss_family* (Z. 6) durchgeführt. Eine dazugehörige Definition einer Struktur/ Union kann auch an dieser Stelle nicht gefunden werden, sodass das Eintreten der umschließenden Bedingung nicht übersetzbaren Quellcode erschafft.

```

1 #ifdef DEBUG
2     fprintf(__stderrp, "New_mmap_list_entry%d(hash_was%d)\n", i, h);
3 #endif
4 ...
5 #ifdef INET6
6     #ifdef WHEN_DOES_THIS_APPLY
7         if ((s->__ss_family == 30) && ...)
8     #endif
9 #endif

```

Abbildung 5.3: Manuelle Fehleranalyse - fehlende Zuordnung 1

Weiterhin zeigt das folgende Beispiel in der Zeile 7 den als Variable identifizierten Bezeichner *GUNZIP*, deren Aufruf auf keine Deklaration zurückgeführt werden kann und somit einen Typfehler verursacht (vgl. Abbildung 5.4).

```

1 #ifdef GUNZIP
2 ...
3     execl(GUNZIP, GUNZIP, "--stdout", "--decompress",
4           req->pathname, ((void *)0));
5 #else

```

Abbildung 5.4: Manuelle Fehleranalyse - fehlende Zuordnung 2

Als drittes Beispiel wird in Abbildung 5.5 ein Auszug aus der Datei *strutil.c* verwendet. Die Funktionsaufrufe von *strchr* in Zeile 1 und *strncmp* in Zeile 2 können keiner Definition/ Deklaration zugeordnet werden. In der Datei sind weitere solcher Aufrufe enthalten, wodurch auf eine Menge von Typfehlern geschlossen werden kann. Eine Abhängigkeit für die Quellcodebereiche besteht in allen Varianten von *boa*, worin die Makros *HAVE_STRDUP* beziehungsweise *HAVE_STRSTR* nicht definiert werden.

```

1 #ifndef HAVE_STRSTR
2 ...
3     for (p = s1; (p = strchr(p, *s2)) != NULL; p = strchr(p + 1, *s2)) {
4         if (strncmp(p, s2, len) == 0)
5             return (p);
6     }
7     ...
8 #endif

```

Abbildung 5.5: Manuelle Fehleranalyse - fehlende Zuordnung 3

Test durch manuelle Konfiguration

Die bisher in diesem Abschnitt aufgeführten Elemente können durch *TypeChef* nicht zurückverfolgt werden. Mit Hilfe der manuellen Konfiguration und anschließender Übersetzung der angepassten Dateien wird die Korrektheit der Fehlererkennung bewertet:

Fehlende Zuordnung 1: Sobald das Merkmal *DEBUG* ausgewählt wird, bricht der Übersetzungsvorgang auf Grund der Variable *h* ab und bestätigt die Inkorrektheit des Quellcodes. Der Komponentenzugriff auf *__ss_family* ist präsent, wenn

die Makros `INET6` und `WHEN_DOES_THIS_APPLY` definiert sind. Daraus lässt sich für `boa` ableiten, dass alle Varianten mit `DEBUG` und der Kombination aus `INET6` und `WHEN_DOES_THIS_APPLY` nicht übersetzt werden können und somit nicht funktionsfähig sind.

Fehlende Zuordnung 2: Mit dem Aktivieren des `GUNZIP`-Merkmals wird der von *TypeChef* erkannte Fehler nicht hervorgerufen. Ausgehend von den untersuchten Dateien sind die Großschreibung des Namens und dessen Abfrage in der umschließenden Direktive ein Anzeichen für eine nicht ausgeführte Makroexpansion (vgl. Abbildung 5.3). Die Textersetzung ist eine Aufgabe des partiellen C-Präprozessors. In Kapitel 6.2 wird diese Fehlersituation genauer untersucht.

Fehlende Zuordnung 3: Die Übersetzung ohne die beiden Makros (`HAVE_STRDUP` und `HAVE_STRSTR`) kann die Menge von Konfigurationsfehlern nicht bestätigen. Die Funktionsaufrufe des originalen und des expandierten Dokumentes können allerdings auch manuell nicht zugeordnet werden, da in dieser Hinsicht keine Inklusionen vorgenommen werden. Eine nähere Betrachtung weist die Elemente als Standardfunktionen aus, die der GCC automatisch bereitstellt und folglich keine Einfügeoperation ausgeführt werden muss.

5.5.2 Typfehler durch Fehlkonfiguration

Die zweite, größere Menge an Typfehlern wird durch den CSP-Solver bei der Analyse der annotierenden Konditionen herausgestellt. Das erste Beispiel beinhaltet die Deklaration der Funktion `_dmalloc_strlen`, die der Bedingung $(\neg \text{DMALLOC_DISABLE} \wedge \text{DMALLOC_FUNC_CHECK})$ (Z. 1, 2) zugeordnet ist (vgl. Abbildung 5.6, Z. 3). In den Zeilen 7 und 8 wird jeweils ein Aufruf der Funktion durchgeführt. Jedoch ist der Inhalt der beiden Zeilen immer vorhanden. Die dazugehörige Deklaration ist nur in Varianten präsent, in denen die oben genannte Bedingung erfüllt wird.

```

1  #ifndef DMALLOC_DISABLE
2      #ifdef DMALLOC_FUNC_CHECK
3      ...
4          extern size_t _dmalloc_strlen(const char * str);
5      ...
6      #endif
7  #endif
8      fakelen = _dmalloc_strlen(fakename);
9      reallen = _dmalloc_strlen(realname);
10 ...
11 #ifdef TEST
12     int fprintf(FILE * , const char * , ...) ;
13 #endif
14 ...
15 if (a < (1<<5)) {
16     fprintf(stderr, "NEEDS_ESCAPE_SHIFT_configuration_error--"
17               "%d should be <= log2(%d)\n", 5, a);
18 }

```

Abbildung 5.6: Manuelle Fehleranalyse - Konditionenvergleich 1

Im zweiten Teil der Darstellung (Z. 10-18) wird die Funktion *fprintf* in jeder Konfiguration aufgerufen, obwohl die passende Deklaration (Z. 11) nur eingeschränkt nutzbar ist. Im Anschluss an ein weiteres Beispiel werden die Varianten manuell nachgestellt und analysiert.

Die Abbildung 5.7 zeigt die Definition einer Union (Z. 2-7), die vollständig von der Abwesenheit der Makros *YYSTYPE* und *YYSTYPE_IS_DECLARED* beeinflusst wird. Die Nutzung der innewohnenden Komponente wird in Zeile 11 mittels des Konstruktes *.cval* ausgeführt. Für das untersuchte Quellcodedokument ist diese Paarbildung eindeutig und wird durch *TypeChef* in ein CSP transformiert. Das Ergebnis ist auf Grund der nicht übereinstimmenden CPP-Direktiven eine fehlerhafte Elementkombination und wird im Rahmen der manuellen Kontrolle bestätigt.

```

1  #if !defined YYSTYPE && !defined YYSTYPE_IS_DECLARED
2      typedef union YYSTYPE
3      {
4          char * sval;
5          int ival;
6          struct ccommand * cval;
7      }
8  #endif
9  ...
10 #ifdef YYPARSE_PARAM
11     if ((yyvsp[(1) - (1)].cval)->action)
12 #endif

```

Abbildung 5.7: Manuelle Fehleranalyse - Konditionenvergleich 2

Test durch manuelle Konfiguration

Auch für diese Menge an Typfehlern wird anhand der betrachteten Beispiele eine manuelle Konfiguration und Übersetzung der Variante durchgeführt, um die Ergebnisse zu bestätigen oder mögliche Fehler von *TypeChef* abzuleiten.

Konditionenvergleich 1: Die beiden Fehlkonfigurationen, aus Abbildung 5.6, können nicht mittels manueller Makrobelegung rekonstruiert werden. Jedoch können beide Situation, speziell deren Entstehen in *TypeChef* nachvollzogen werden.

Die Untersuchung des ersten Quellcodeabschnittes in der unveränderten sowie expandierten C-Datei zeigt als Ursache die variable Definition von Makros, was eine gebräuchliche Methode bei der CPP-Konfiguration darstellt. Hierbei geschieht die Expansion ausschließlich in bestimmten Varianten oder eine spezifische Festlegung des Ersatztextes. Im Beispiel wird der Funktionsaufruf in einer gewissen Menge von Varianten ersetzt. Der partielle C-Präprozessor löst die Makrodefinitionen und dazugehörigen Inklusionen (benötigte Funktionsdeklarationen) jedoch aus diesem Kontext heraus, sodass die Expansion für jedes Auftreten durchgeführt wird. Abschließend wird der Variabilitätskontext reintegriert. In dem gezeigten Beispiel resultiert der immer präsente Zugriff auf eine annotierte Funktion.

Das untere Beispiel nutzt die Funktion *fprintf* aus der Bibliothek *stdio.h*. Die Fehlerrekonstruktion leitet sich von einer konfigurierten Präprozessoranweisung, insbesondere der Inklusion einer Header-Datei, ab. Der partielle C-Präprozessor löst die alternativen **#include**-Hierarchien auf. Die benötigte Funktionsdeklaration wird in beiden Konfigurationen bereitgestellt. Die dazugehörige Datei ist mit einem Einfügeschutz versehen und verhindert eine mehrfache Ausführung. In dem betroffenen Abschnitt wird die Funktion in beiden Alternativen zugänglich. Die Arbeitsweise von *TypeChef* führt beide Direktiven aus, jedoch wird der Inhalt nur im ersten Kontext eingefügt. Dementsprechend entsteht eine Referenzbeziehung, die nicht in allen Varianten hergestellt werden kann.

Konditionenvergleich 2: Das Referenzpaar in Abbildung 5.7 ruft einen Typfehler hervor. Die parametrisierte Übersetzung kann den durch *TypeChef* angezeigten Fehler nicht direkt bestätigen. Die für die betroffene Konfiguration angepasste Variante kann dennoch nicht übersetzt werden, da der resultierende Quellcode durch das Fehlen einer schließenden Klammer als invalide anzusehen ist. Der Transformationsfehler tritt bereits vor dem gesuchten Bereich auf und verursacht eine inkorrekte Struktur im Dokument.

5.5.3 Bestätigte Fehler

Die manuelle Kontrolle der Ergebnisse kann für fehlerhaft gekennzeichnete Referenzpaare die Existenz nachweisen. Daraus leiten sich inkorrekte Konfigurationen ab, die durch *TypeChef* ermittelt werden konnten. Die Auswahl von bestimmten Merkmalen ruft Inkonsistenzen hervor, sodass die Übersetzung des Quellcodes nicht vollzogen werden kann.

Über das Makro **DEBUG** werden Ausgaben in die C-Dateien integriert, die den Programmablauf dokumentieren. Eine nicht zurückverfolgbare Variable wird durch die Transformation in den Quellcode eingefügt, wodurch alle Varianten von *boa* mit diesem Merkmal nicht übersetzbar sind.

Durch die Definition **INET6** und **WHEN_DOES_THIS_APPLY** wird ein Zugriff auf die Komponente `__ss_family` integriert, obwohl keine Struktur/ Union einen solchen Bestandteil besitzt. Dieser Fehler konnte durch die Auswahl der beiden Merkmale rekonstruiert werden.

Eine ähnliche Situation wird durch **YYPARSE_PARAM** verursacht. Die Selektion des Makros offenbart Syntaxfehler in den resultierenden Konfigurationen. Die von *TypeChef* beschriebene Fehlerzusammensetzung entsteht nicht, wenn sich die wirkenden Merkmale für die obige Definition und den Komponentenzugriff ausschließen (vgl. Abbildung 5.7). Die Anwendung *boa* besitzt kein Merkmalmodell und eine Ableitung des Modells ist nicht möglich. Folglich werden die unterschiedlich markierten Elementoperationen als Typfehler herausgestellt, da keine Abhängigkeiten zwischen den Merkmalen festgelegt werden können.

Die Überprüfung der Funktionsweise konnte mehrere problematische Quellcodekonstrukte aufzeigen, die eine zusätzliche Erweiterung des partiellen C-Präprozessors benötigen. Die damit verbundenen Probleme werden anschließend in Kapitel 6 aufgegriffen sowie mögliche Lösungsstrategien betrachtet.

5.6 Unterstützte und analysierbare CPP-Konditionen

Der C-Präprozessor erlaubt die Formulierung von Konditionen, die über die Abfrage der An- oder Abwesenheit eines Makros hinausgehen. Der CSP-Transformator in *TypeChef* unterstützt CPP-Direktiven, die komplexe Bedingungen zwischen Makros und Ganzzahlkonstanten festlegen. In *boa* wird ausschließlich die Existenz einer solchen Konfigurationsvariablen genutzt, um die Präsenz eines Elementes zu beschreiben.

Nachdem im Rahmen der ersten Fallstudie gezeigt werden konnte, dass *TypeChef* in der Lage ist, Typfehler in einer bestehenden Anwendung zu finden, wird die Analysierbarkeit von anderen Konditionen untersucht. Das Ziel dieses Abschnittes ist die Untersuchung von *gretl*², da für die Implementierung eine höhere Komplexität in der Formulierung der CPP-Bedingungen verwendet wird und das Transformationssystem in *TypeChef* in der Lage ist, diese Ausdrücke bei der Typfehlerprüfung auszuwerten.

Für das verfolgte Vorhaben wird ein Dokument aus dem Quellcode von *gretl* entnommen. Die darin enthaltenen Annotationen sind ausreichend, um verschiedene Bedingungen und die Verarbeitung des sich ergebenden CSP aufzuzeigen. Anhand des Dokumentes *bootstrap.c* wird im weiteren Verlauf des Abschnittes erläutert, inwiefern relationale Aussagen zur Reaktion auf mehrere Ebenen der Programmdiagnose genutzt und durch *TypeChef* analysiert werden können. Das bewertete Makro *BDEBUG* legt hierbei drei Ebenen fest und löst eine spezifische Ablaufdokumentation aus. Je nach Existenz von *BDEBUG* werden Ausgaben über das Verhalten integriert (vgl. Abbildung 5.8, links). Des Weiteren werden die Diagnoseabschnitte mit detaillierteren Informationen versehen, wenn *BDEBUG* den Wert 1 beziehungsweise einen höheren Wert annimmt (vgl. Abbildung 5.8, rechts).

<pre> 1 #if BDEBUG 2 fprintf(stderr, ...); 3 gretl_matrix_print(R, "R"); 4 gretl_matrix_print(q, "q"); 5 #endif </pre>	<pre> 1 int flags = 0; 2 ... 3 #if BDEBUG > 1 4 flags = BOOT_VERBOSE; 5 #endif </pre>
--	---

Abbildung 5.8: Relationale Konditionen (*gretl*)

TypeChef kann keine Fehler herausstellen, die zu inkonsistenten Konfigurationen führen. Ein Großteil der Referenzpaare muss nicht untersucht werden, da die erschaffende Operation, wie die Variablendeklaration in Zeile 2 des rechten Beispiels, nicht annotiert und somit immer erreichbar sind. Für fünf Konstellationen ist der Test notwendig. Die Untersuchungen betreffen einen Bereich, der mit Hilfe der gleichen Bedingung beide Elementoperationen variabel anpasst.

Transformation von komplexen Direktiven

Der im rechten Teil der Abbildung 5.8 enthaltene Abschnitt wird als Grundlage verwendet, um die Funktionsweise der CSP-Transformation mittels manuell konstruierter Annotationen aufzuzeigen. Hierbei wird das bestehende Paar verschiedenartig mar-

² <http://gretl.sourceforge.net/>

kiert und Bedingungen erschaffen, die in *TypeChef* in ein CSP überführt und durch den Solver auf Erfüllbarkeit getestet werden (vgl. Abbildung 5.9).

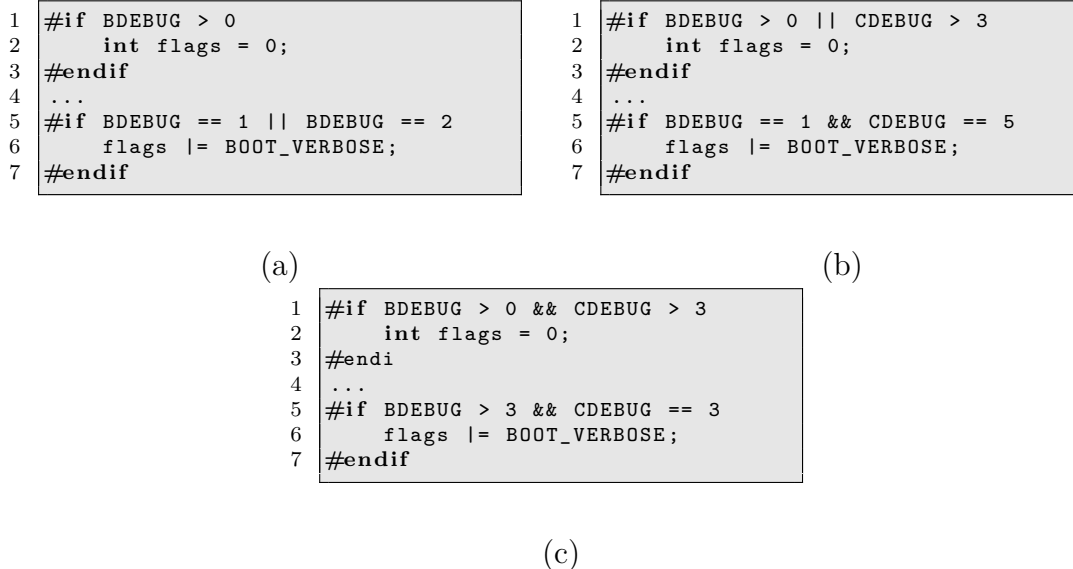


Abbildung 5.9: Weitere durch *TypeChef* untersuchbare Konditionen

Das Beispiel (a) verschiebt die Deklaration in alle Varianten, in denen `BDEBUG > 0` gilt. Der Zugriff ist auf genau zwei Fälle eingegrenzt, aber nur dann vorhanden, wenn eine der mit oder verknüpften Aussagen zutrifft (Z. 5). Der CSP-Solver bestätigt die Korrektheit der Annotation.

Die weiteren Paarungen in (b) und (c) zeigen die zusätzliche Abhängigkeit von einem hier neu eingeführten Makro *CDEBUG*. Die Bedingungen für die Präsenz von Deklaration und Zugriff formulieren relationale Abhängigkeiten der Makros von Ganzzahlkonstanten und logische Verknüpfungen zwischen den Teilaussagen. Beide Beispiele können durch *TypeChef* untersucht werden. Für den Abschnitt (b) werden in der Solver-Phase keine Belegungen der beiden Makros ermittelt, um die intern konstruierte Typfehleraussage zu erfüllen. Die letzte Konstellation (c) ruft einen Typfehler hervor. Die Abhängigkeit von *BDEBUG* ist bei der Nutzung der Variable *flags* eingeschränkter formuliert und somit nicht ausschlaggebend. Der CSP-Solver findet mit *CDEBUG* = 3 eine Belegung, in der für den Zugriff keine passende Deklaration im Quellcode vorhanden ist.

Die gezeigten CPP-Direktiven konstruieren unterschiedliche Aussagen für die Präsenz der beteiligten Elemente. Neben den relationalen und logischen Verknüpfungen ist *TypeChef* in der Lage, arithmetische Gleichungen als Bestandteile einer der oben dargestellten Aussagen aufzuschlüsseln und in die Beschreibung eines CSP aufzunehmen. Die Beispiele dienen der Veranschaulichung der zulässigen Anweisungen, wobei diese verschiedenartig kombiniert und zur Konstruktion einer komplex verschachtelten Hierarchie verwendet werden können.

5.7 Kapitelzusammenfassung

Das entwickelte Werkzeug *TypeChef* analysiert die Referenzen in dem bestehenden C-Quellcode unter Betrachtung der CPP-Abhängigkeit auf das Hervorrufen von Typfehlern. Für die Anwendung *boa* wurde die vollständige Untersuchung durchgeführt. Die Ergebnisse von *TypeChef* werden aus den expandierten Dokumenten des partiellen CPP extrahiert, sodass die hierbei herausgestellten Fehler über die Inklusionshierarchie wiederholt in die C-Dateien einbezogen werden. Für ausgewählte Referenzpaare wurde die Konfiguration nachgestellt, die die Analyse als inkonsistent gekennzeichnet hat. Zwei der verwendeten Makros verursachen die Generierung einer fehlerhaften und nicht übersetzbaren Variante des Anwendungsquellcodes. Weiterhin wird durch eine Kombination aus `INET6` und `WHEN_DOES_THIS_APPLY` eine Strukturveränderung herbeigeführt, die der Compiler abweist.

Die Durchführung der ersten Fallstudie, speziell die manuelle Kontrolle der Ergebnisse, zeigt die korrekte Funktionsweise der Bestandteile zur Ermittlung der Referenzen und die Betrachtung der Konditionen in Form eines CSP. Ein Großteil der nicht rekonstruierbaren Fehler wird durch Grenzen des partiellen CPP erschaffen, die mit Hilfe der Evaluation identifiziert wurden. Die technischen Grenzen von *TypeChef* werden in Kapitel 6 betrachtet und mögliche Perspektiven vorgestellt, die in zukünftigen Arbeiten genauer untersucht werden.

In der zweiten Fallstudie wurde ein Dokument aus dem Projekt *gretl* entnommen, da die Variabilität des Quellcodes mit Makrogleichungen realisiert wird, die durch den CPP formuliert und seitens *TypeChef* verarbeitet werden. Anhand einer Beispielpaarung aus Variablendeklaration und -zugriff werden CPP-Konditionen gezeigt und auf Typfehler untersucht. Die verschiedenen Anpassungen legen den Funktionsumfang von *TypeChef* für die im Rahmen der bedingten Übersetzung erlaubten CPP-Anweisungen dar.

Das anschließende Kapitel greift Probleme und Grenzen von *TypeChef* erneut auf und beschreibt Strategien zu Verbesserung sowie Erweiterung des Werkzeuges.

KAPITEL 6

Probleme und Perspektiven

Die Untersuchung der Einflussnahme von Präprozessoranweisungen auf die Korrektheit einer gesamten SPL und somit aller Varianten einer Anwendung ist eine komplexe Thematik. Ein Ziel der Arbeit ist es, herauszustellen, inwiefern ein produktlinienorientiertes Typsystem für C und der Annotationstechnik des CPP erschaffen werden kann.

Das Erreichen der Vollständigkeit des hierbei entstehenden Werkzeuges konnte auf Grund der weitreichenden Zusammenhänge bei der Variabilität des C-Präprozessors im Rahmen dieser Arbeit nicht realisiert werden. Dementsprechend liegt der Fokus grundlegend auf der Erschaffbarkeit von *TypeChef*. Auf der Basis einer prototypischen Umsetzung wird weiterhin das Ziel verfolgt, Grenzen und Probleme bei der Umsetzung eines Werkzeuges für die statische Referenzanalyse in C aufzuzeigen.

Die Entwicklung von *TypeChef* und speziell die praktische Anwendung auf das bestehende Projekt *boa* haben ebensolche Probleme und Einschränkungen hervorgebracht.

BESSEY et al. [BBC⁺10] beschreiben die Entwicklung eines kommerziellen Werkzeuges zur Analyse von C-Projekten. Hierbei werden spezielle Aspekte und Erfahrungen beleuchtet. Ein wichtiger Punkt ist die Abhängigkeit des Untersuchungsobjektes vom zur Erstellung genutzten Compiler. Wenn diese veraltet sind oder nicht dem Standard entsprechen, werden Analysesysteme vor besondere Aufgaben gestellt, da abgewandelte oder spezialisierte Sprachkonstrukte entstehen, die in einem allgemeinen C-Werkzeug nicht untersucht werden können oder zu falsch positiven Ergebnissen führen [BBC⁺10]. Die Auswirkung des Compilers auf die Konformität der Sprachkonstrukte ist nur einer von vielen Aspekten, die das Erreichen der Vollständigkeit eines Analysesystems für C verhindern.

Das vorliegende Kapitel dient der detaillierten Betrachtung der verschiedenen technischen Grenzen, die erst durch die Untersuchung einer reellen Anwendung hervortreten, sowie der Vorstellung von möglichen Lösungsperspektiven. Ausgehend von nicht nachvollziehbaren Elementen werden Verbesserungen des partiellen C-Präprozessors und der Referenzanalyse erläutert.

6.1 Nicht nachvollziehbare Elemente

Die Analyse von *boa* kann eine gewisse Anzahl von Referenzen nicht dem dazugehörigen Element zuordnen, da die Symboltabelle keine passenden Informationen beinhaltet. In der Menge der offenen Verbindungen wird ungefähr jedes dritte Element durch das C-Konstrukt *Label* verursacht. Basierend auf Traversierung des AST und

der parallel realisierten Paarbildung können keine Sprungmarken verarbeitet werden, da das Ziel des Sprunges erst in einem tieferen Teil des Baumes lokalisiert wird. Jedoch existieren verschiedene Möglichkeiten, um die Analyse eines Labels durchzuführen.

Mögliche Verarbeitungsansätze

Die in *TypeChef* bereits enthaltenen Mechanismen lassen sich hinsichtlich der Referenzanalyse erweitern oder durch Restrukturierung des Anwendungs Quellcodes direkt verwenden:

Nachträgliche Paarbildung: Die verwendete Grammatik kapselt den Bezeichner eines Labels durch den Pseudoknoten *NLabel*. Der dazugehörige Aufruf wird in *TypeChef* bereits als Variablenzugriff erkannt. Folglich kann während der Traversierung eine separate Tabelle für die Marken erschaffen werden. Nach dem Abschluss der Untersuchungen werden dann alle Variablen ohne Zuordnung mit dieser Tabelle auf mögliche Zusammenhänge verglichen. Die dabei entstehenden Paare werden im bestehenden Ablauf von *TypeChef* auf Typfehler untersucht.

Kapselung des Label-Inhaltes: Die Sprungmarke definiert den Beginn eines Verarbeitungsabschnittes. Durch die Restrukturierung des Bereiches wird der Quellcode in eine Funktion gekapselt und vor der aktuellen Funktion mittels der spezifischen Operation definiert. Dementsprechend ist jede Nutzung eines Sprunges zu ermitteln und durch einen Aufruf der extrahierten Funktion zu ersetzen. Die rekursive Anwendung des Mechanismus erfasst zusätzlich die Schachtelung von Marken. Die Restrukturierung erlaubt die unveränderte Nutzung der bisherigen Referenzanalyse, da das Element Funktion und die Operation verarbeitet werden. Die erläuterte Strategie ist jedoch mit weiteren Nachteilen verbunden. Alle in einem Label genutzten Elemente, wie lokale Variablen oder Definitionen von eigenen Typen, müssen in die jeweils erstellte Funktion übergeben werden oder auf die globale Ebene verschoben werden. Hierbei sind weitere Untersuchungen notwendig, die die Entstehung von Mehrdeutigkeiten verhindern.

Mit Hilfe der Label wird eine gewisse Anzahl von Elementen abgedeckt, die in *TypeChef* keine Zuordnung erhalten. Die manuelle Kontrolle der verbleibenden Menge kann keine Fehler in der Referenzermittlung feststellen. Eine detaillierte Analyse der einzelnen Elemente identifiziert zusätzlich folgende Ursachen:

Fehlerhafter Untersuchungs Quellcode: Aus dieser Situation lässt sich generell auf einen fehlerhaften Quellcode in den betroffenen Dokumenten schließen. Die Anwendung wird mittels des CPP hinsichtlich verschiedener funktioneller Merkmale angepasst. Folglich können die Fehler ein Bestandteil von unbenutzten Varianten sein und immer aus der Quellcodebasis entfernt werden, obwohl *TypeChef* die Gesamtbasis als inkonsistent einstuft.

Auflösung von Direktiven mit Konfigurationskontext: Die Menge der nicht nachvollziehbaren Elementzugriffe besteht aus jeglicher Art von vorhandenen Typen. Eine genauere Untersuchung dieser Elemente hinsichtlich der Gestalt der Benennung legt den Schluss nahe, dass an dieser Stelle keine Makroersetzung durchgeführt wurde, da der vorgefundene Bezeichner ebenso in den kapselnden Direktiven genutzt wird. Demnach wird das Makro nicht definiert und der fehlerhafte Quellcode nicht erreicht. Das entwickelte Analysewerkzeug ignoriert die Abstinenz des Makros und verfolgt alle generierbaren Konfigurationen.

Die erweiterten, jedoch manuellen Untersuchungen weisen weiterhin die Existenz von kontextabhängigen Makrodefinitionen in der Fallstudie nach. In Abhängigkeit von gewählten Funktionalitäten, werden Makros mit verschiedenen Versionen des Ersatztextes versehen und bewirkt die Referenzierung von nicht ermittelbaren Elementen. Die Einflussnahme des partiellen CPP wird im folgenden Abschnitt gesondert betrachtet, da daraus mehrere Probleme abgeleitet werden können.

6.2 Partieller C-Präprozessor

Die partielle Ausführung des C-Präprozessors stellt die Quelle für eine Vielzahl der nicht nachvollziehbaren Zugriffe dar und ruft weiterhin Konfigurationsfehler hervor, die von der innewohnenden Arbeitsweise verursacht werden. Bereits im Rahmen der Fallstudie werden verschiedene Gesichtspunkte dieser Problematik aufgeführt. Im Verlauf dieses Abschnittes werden die ermittelten Einschränkungen diskutiert und Ansätze für Lösungen skizziert.

6.2.1 Plattformspezifische Konfiguration

Die Referenzanalyse eines C-Projektes baut auf der korrekten Verarbeitung des partiellen CPP auf. Ein Problemfeld in dieser Phase ist die plattformspezifische Zusammensetzung des Quellcodes. Die Anpassung an mehrere Zielsysteme nutzt Standarddateien, die der GCC nicht bereitstellt. In *TypeChef* werden die hierfür genutzten Makros durch einen prototypischen Mechanismus vor dem Auskommentieren geschützt. Die resultierenden CPP-Direktiven werden durch den GCC ausgewertet und es wird entschieden, dass die Inklusionen nicht existieren und die dazugehörigen Änderungen nicht durchgeführt werden können. Für umfangreichere und geschachtelte Direktiven müssen weitere manuelle Anpassungen vorgenommen werden.

6.2.2 Konfiguration von Direktiven

Die vom CPP ausgeführte Texttransformation ist nicht auf den C-Quellcode begrenzt. Wie auch in der Fallstudie *boa* enthalten, werden auch Anweisungen, die die Makrodefinitionen oder Dateiinklusionen realisieren, für die Varianten des Projektes unterschiedlich formulieren. Diesbezüglich werden die Direktiven `#define` und `#include` sowie die Beeinflussung mit Hilfe von Annotationen näher beleuchtet.

Bedingte Makrodefinitionen

Speziell für die Menge von Makrodefinitionen werden alle Vorkommen der umzuschreibenden Zeichenketten ersetzt, obwohl eine im Originaldokument vorhandene Annotation den Bereich für diese Transformation einschränkt. Folglich können die Zusammensetzung einer Quellcodedatei und der enthaltene Kontext sehr stark verändert werden. Eine weitere Fehlersituation kann entstehen, wenn Makros abhängig von der Konfiguration definiert werden. Die Auskommentierungsphase erzeugt mehrfache Definitionen, die der GCC als Redefinition interpretiert, sodass immer das letzte Muster als Grundlage bei der Textersetzung eingesetzt wird.

Bedingtes Einfügen von Dateiinhalten

Über die bedingte Inklusion wird durch den CPP entschieden, in welcher Konfiguration bestimmte Dateien genutzt werden. Die aktuelle Umsetzung des partiellen Präprozessors löst die `#include`-Hierarchie unabhängig vom Kontext auf. Falls in den alternativen Blöcken gleiche Dateien verwendet werden, verbleibt der darin inkludierte Quellcode im Kontext der ersten Direktive oder wird mehrfach eingefügt, wenn keine Include Guards vorhanden sind. In der Fallstudie *boa* werden fehlerhafte Referenzpaare durch das Einfügen von Dokumenten erzeugt, die in allen Varianten verfügbar sind, jedoch in einer anderen Hierarchie verarbeitet werden.

Das beschriebene Verhalten ist auch auf jegliche andere Direktive anwendbar, wonach beispielsweise eine Makrodefinition zu früh gelöscht wird und daraus nicht nachvollziehbare Elemente resultieren. Aus dem Umfang der bedingt gestalteten Direktiven lässt sich ableiten, inwiefern die Analyseergebnisse verändert und somit möglicherweise verfälscht werden können.

6.2.3 Perspektiven des partiellen Präprozessors

Der bestehende Mechanismus zur Erkennung und Auflösung von systembezogenen, bedingten Anweisungen ist auf komplexe Konstrukte zu erweitern. Weiterhin sind Auswirkungen auf den Konfigurationskontext zu beachten und involvierte Annotationen gegebenenfalls umzuformen, um die Konsistenz der Anwendung zu bewahren. Speziell bei der kombinierten Abfrage von System- und Konfigurationsmakros muss die von *TypeChef* zu untersuchbare Variabilität extrahiert werden.

Die variablen Anpassungen der beiden Direktiven `#define` und `#include` sind Quellen für die fehlerhafte Transformation und wirken sich demnach negativ auf die Analysen von *TypeChef* aus. Das verbesserte Auflösen dieser speziellen Konstellationen ist notwendig, um die Quellcodebasis für alle Varianten korrekt zu erschaffen.

Ein möglicher Ansatz ist die Realisierung eines produktlinienorientierten C-Präprozessors. Hierbei wird der vorhandene CPP ersetzt, wodurch der Funktionsumfang des Werkzeuges vollständig nachgebildet werden muss, aber auf die besonderen Anforderungen anpassbar ist.

Bei der Expansion eines Makros werden gleichzeitig die Konfigurationsbedingungen beachtet. Der Mechanismus erschafft den Kontext einer Makrodefinition auch an der

Position jeder Ersetzung und expandiert das Makro in den verschiedenen Varianten. Somit wird der jeweilige Abschnitt zwar dupliziert, aber eine fehlerhafte Verarbeitung ausgeschlossen.

Das bedingte Einfügen von Dateiinhalten in einem eigenständigen Präprozessor bedarf einer zusätzlichen Analyse der CPP-Hierarchie eines untersuchten Dokumentes. Für die Inklusion und alle weiteren Direktiven wird an dieser Stelle keine Lösung beschrieben, da gesondert zu untersuchen ist, welchen Einfluss die variable Umrahmung hat beziehungsweise inwiefern der Kontext mit Hilfe eines produktlinienorientierten CPP folgerichtig erhalten werden kann.

Der in *TypeChef* verwendete, partielle Präprozessor löst grundlegend schwierige Probleme. Jedoch erlaubt die Annotation eine hohe Komplexität der im Fokus stehenden CPP-Direktiven, die erst mit Hilfe der Fallstudie herausgestellt werden konnten. Bedingte sowie plattformspezifische Präprozessoranweisungen stellen eine Hürde für *TypeChef* dar und beeinflussen die Referenzanalyse. Die skizzierten Ansätze können in diesem Zusammenhang eine Korrektur und somit eine Verbesserung hervorrufen.

6.3 Alternativer C-Quellcode

Die Beeinflussung der Analyse gilt auch für Annotation des Quellcodes durch alternative Präprozessoranweisungen. In den umschlossenen Blöcken wird die Definition/Deklaration von Elementen mit einem identischen Bezeichner vollzogen und somit mehrfach bereitgestellt. In dem entwickelten Analysewerkzeug wird keine gesonderte Verarbeitung durchgeführt. Dementsprechend wird in der Phase der Paarbildung die mehrfache Existenz des erschaffenden Elementes registriert, alle Konstellationen gebildet und durch den CSP-Solver untersucht. Das Lösungssystem testet jede Nutzung des Elementes gegen alle Definitionen/ Deklarationen, wodurch die Fehlerfreiheit eines Paares als ausreichend festgelegt wird.

TypeChef ist dahingehend zu erweitern, dass für jede Alternative zusätzlich die mehrfach vorhandenen Definitionen/ Deklarationen gegenübergestellt werden. Hierbei werden die Annotationen der Elemente ausgelesen und eine Fehlersituation konstruiert, die die gleichzeitige Existenz der Operationen formuliert. Kann für alle Alternativen kein Fehler ermittelt werden, ist in jeder Konfiguration ausschließlich eine Definition/ Deklaration enthalten.

Diese Untersuchung ist in die Phase der Paarbildung zu integrieren. Für die Prüfung kann der CSP-Transformationsmechanismus verwendet werden. Die resultierenden Bedingungen werden konjugiert und insgesamt negiert. Wenn das Lösungssystem Choco keine Variablenbelegung herausstellen kann, ist in jeder Variante nur eine der erschaffenden Operation vorhanden. Die Einführung des Tests korrigiert die Referenzanalyse und ermöglicht die Verarbeitung von alternativ konfigurierten Definitionen sowie Deklarationen.

6.4 Perspektiven für ein produktlinienorientiertes Typsystem

Die in *TypeChef* durchgeführten Überprüfungen vollziehen die statische Referenzanalyse für alle konstruierbaren Varianten eines Projektes. Für die verschiedenen Operationen werden alle zur Analyse benötigten Informationen extrahiert. Die Paarbildung resultiert aus der Art des Elementes und dem Bezeichner.

Ein zusätzlicher Schritt ist die Einführung von Typprüfungen für die zusammengestellten Paare. Folglich ist die Referenzanalyse dahingehend zu erweitern, dass für die einzelnen Elemente Typinformationen gesammelt und in der Symboltabelle hinterlegt werden. Bei der Bildung eines Paares kann in Ergänzung zur Sicherstellung der variablen Verfügbarkeit eines Elementes auch die Typkorrektheit herausgestellt werden. Die Prüfungen können beispielsweise für einen Funktionsaufruf ermitteln, dass zusätzlich zur CPP-Variabilität die Typen der als Parameter übergebenen Elemente für alle Quellcodevarianten korrekt sind.

Mit Hilfe von Typuntersuchungen wird eine Begrenzung von *TypeChef* aufgelöst. Die Verarbeitung von Komponentenzugriffen wird auf Grund der Verarbeitung nicht korrekt ausgeführt, wenn mehrere Komponenten mit dem gleichen Bezeichner existieren. Die Durchführung einer Typprüfung erlaubt das Auflösen des Kontextes, in dem die Komponente referenziert wird. Hierbei ist der Zugriff auf die beinhaltende Struktur/ Union zurückzuführen, um die Herkunft des Elementes eindeutig zu bestimmen. Daraus folgt die korrekte Verknüpfung zwischen der Definition und der Verwendung der Komponente. *TypeChef* wird demnach in die Lage versetzt, auch diese Paare zu analysieren und die Konditionen für die Präsenz des Elementes für alle generierbaren Quellcodevarianten zu untersuchen.

6.5 Kapitelzusammenfassung

Die Entwicklung von *TypeChef* und die Fallstudie *boa* haben verschiedene Probleme der statischen Referenzanalyse aufgezeigt. Nicht nachvollziehbare Elemente lassen sich auf nicht verarbeitete Label der Sprache C und die partielle Ausführung des CPP zurückführen. In diesem Zusammenhang werden plattformspezifische Inklusionen sowie die variable Gestaltung von Präprozessordirektiven als Grenzen von *TypeChef* aufgezeigt. Dadurch wird ein Ziel der Arbeit erfüllt und Probleme bei der Realisierung eines produktlinienorientierten Typsystems ermittelt. Die untersuchten Grenzen und die unvollständige Analyse von Alternativen stellen Perspektiven für die Erweiterung von *TypeChef* dar. Abschließend werden die Vorteile von zusätzlichen Typprüfungen und der damit verbundenen Aufhebung einer Begrenzung während der Paarbildung beleuchtet.

KAPITEL 7

Verwandte Arbeiten

Das vorliegende Kapitel dient der Abgrenzung zu anderen Arbeiten. Hierbei werden Werkzeuge und Ansätze vorgestellt, die das Zusammenspiel der Direktiven des C-Präprozessors und Quellcode der Programmiersprache C untersuchen. Die Resultate bilden die Grundlage für die Restrukturierung und die Analyse der Korrektheit der erzeugten Variabilität. *TypeChef* realisiert ein produktlinienorientiertes Typsystem basierend auf der statischen Referenzanalyse. Im zweiten Abschnitt des Kapitels werden weitere Ansätze betrachtet, die eine SPL und somit alle generierbaren Varianten vollständig auf Korrektheit überprüfen.

7.1 Analyse von C und CPP

Der folgende Abschnitt beschreibt verschiedene Analysewerkzeuge, die die Beeinflussung der Anwesenheit von Präprozessordirektiven auf den C-Quellcode untersuchen. Die Ansätze sind auf die Restrukturierung sowie die Sicherstellung der Korrektheit der CPP-spezifischen Anpassungen ausgerichtet.

Restrukturierung

Die Analyse des CPP wird im Gebiet der Restrukturierung von bestehenden C-Projekten eingesetzt. Hierbei liegt die Einflussnahme von C-Präprozessoranweisungen auf die Korrektheit der resultierenden Quellcodeveränderungen im Fokus. Die Untersuchungen werden mit Hilfe der Werkzeuge *CRefactory* und *Xrefactory* durchgeführt [GJ03, Vit03, GJ05].

Der Ansatz von *CRefactory* nutzt einen Pseudopräprozessor zur Sammlung von Informationen über die vorhandenen Direktiven. Darauf aufbauend werden Makros expandiert und Annotationen dahingehend verändert, dass diese der vom internen Parser verwendeten Grammatik entsprechen. Der erstellte AST beinhaltet alle Anpassungen an die verschiedenen Konfigurationen und bildet die Grundlage der anschließenden Restrukturierung [GJ03, GJ05].

Das Werkzeug *Xrefactory* verfolgt ein ähnliches Ziel, jedoch werden die durchzuführenden Manipulationen, wie das Einfügen von Dateiinhalten und die Substitution von Makros, direkt auf der Ebene der Quellcodedateien angewendet [Vit03].

Wie auch in der vorliegenden Arbeit wird in beiden Ansätzen der Quellcode intern anhand eines AST verarbeitet. Die Analyse des Baumes wird nicht zur Sicherstellung der Korrektheit aller Konfigurationen verwendet, sondern ist auf die Restrukturierung von C-Anwendungen ausgerichtet.

DMS

In BAXTER und MEHLICH [BM01] wird mit *DMS* ein kommerzielles System für die Transformation des Quellcodes in den Programmiersprachen C und C++ präsentiert. Die Grundlage bildet eine Grammatik, die es ermöglicht, den Quellcode im Kontext der Direktiven des CPP zu analysieren. Hierbei sind die Präprozessoranweisungen in ihrem Auftreten begrenzt und werden nur an bestimmten Stellen zugelassen. Die Vorgehensweise dient der Realisierung eines disziplinierten Einsatzes und der Absicherung der Quellcodemanipulation durch den CPP. Ausgehend von den verwendeten Einschränkungen treffen BAXTER und MEHLICH die Aussage, dass DMS 85% der Dateien einer bestehenden Anwendung verarbeitet und die notwendige Überführung der undisziplinierten Konstrukte „an einem Nachmittag“ vollzogen werden kann [BM01].

Ähnlich wie der in Rahmen der vorliegenden Arbeit verfolgte Ansatz arbeitet DMS mit einer modifizierten Grammatik für C und CPP sowie der Darstellungsform eines AST für die weiterführenden Untersuchungen. Die Sicherstellung der disziplinierten Verwendung von Direktiven ist bereits direkt in der Grammatik hinterlegt. Auch in DMS sind die Disziplinverletzungen durch einen zusätzlichen, manuellen Aufwand zu korrigieren. Der erweiterte AST wird nicht, wie in *TypeChef*, für die Korrektheitsanalysen in den generierbaren Varianten genutzt. DMS nutzt den Baum als Grundlage für die Vereinfachung von C- sowie CPP-Konstrukten, wodurch beispielsweise nicht erreichbare, aber variable Bereiche erkannt werden.

Weiterhin wird in AVERSANO et al. [ADPB02] ein Ansatz skizziert, der das Werkzeug DMS für Typprüfungen im Zusammenhang mit der bedingten Übersetzung und den darin enthaltenen Variablen im C-Quellcode erweitert, um fehlerhafte Konfigurationen zu ermitteln.

ASTEC

MCCLOSKEY und BREWER [MB05] stellen mit *ASTEC* ein Analysesystem vor, das die bedingte Übersetzung in einer disziplinierten Nutzung betrachtet, um die Analysierbarkeit und eine bessere Restrukturierung zu ermöglichen. Bei der Verwendung des Werkzeuges wird der bestehende C-Quellcode in eine Transformation übergeben. Hierbei werden alle im Dokument befindlichen CPP-Direktiven in eine spezielle, eigenständige Makrosprache übersetzt. Die während des Vorganges genutzte Struktur ist auch in diesem Ansatz ein AST. MCCLOSKEY und BREWER argumentieren, dass durch die vorzunehmende Transformation eine Vielzahl der Fehler eliminiert werden, die die generelle Anwendung des CPP hervorruft. Jedoch ist diese Eigenschaft von der folgerichtigen Transformation durch das verwendete Parsersystem abhängig [MB05].

ASTEC unterscheidet sich grundsätzlich durch die Direktivenübersetzung in eine eigene Makrosprache von dem in der vorliegenden Arbeit erschaffenen Werkzeug. Die bedingte Übersetzung wird ausschließlich auf der Ebene von Deklarationen, Anweisungen und Ausdrücken erlaubt. Weiterhin werden die in den Direktiven formulierten Aussagen analysiert und für die Wahl einer Verzweigung genutzt. Folglich wird keine gleichzeitige Untersuchung aller generierbaren Varianten ermöglicht. Ein weiterer Unterschied ist die Art der Sicherstellung von disziplinierten Annotationen. In ASTEC

wird die Einhaltung mittels des Übersetzers forciert und undisziplinierte Markierungen auf die nächst valide Ebene verschoben [MB05].

Im Gegensatz zu *TypeChef* ist ASTEC abhängig von der korrekten Ausführung der vorherigen Transformation des Quellcodes, da die anschließenden Untersuchungen darauf aufbauen. Bei der Transformation wird eine Konfiguration festgelegt, sodass die gesamte SPL nicht analysiert werden kann.

Symbolische Ausführung

Eine weitere Analysemethode wird von HU et al. [HMDL00] vorgestellt. Hierbei wird die Einflussnahme der bedingten Übersetzung mittels der symbolischen Ausführung ausgewertet. Die symbolische Ausführung wird generell genutzt, um Schlussfolgerungen über alle möglichen Eingaben, welche zu einem gemeinsamen Pfad durch die Anwendung gehören, zu treffen. Die Eingaben werden als Symbole interpretiert und ermöglichen die Ausführung eines Programms für mehrere solcher Pfade. Die Ausführungspfade werden baumartig aufgebaut und legen unter Verwendung der Symbole eine Bedingung für jeden Pfad fest, die den Weg durch den Baum beschreibt [Kin76].

HU et al. [HMDL00] nutzen dieses Prinzip für die Untersuchung der bedingten Anweisungen des CPP, wobei die Existenz oder die Werte der verwendeten Makros als symbolische Konstanten für die Analyse von unterschiedlichen Ausführungen, speziell Konfigurationen, des Quellcodes genutzt werden. Die symbolische Ausführung extrahiert einen AST aus dem Dokument. Im Gegensatz zu *TypeChef* enthält der Baum ausschließlich die Anweisungen des CPP, da die verwendete Grammatik nicht für das Parsen von annotiertem C-Quellcode ausgelegt ist. Diesbezüglich liegt der Fokus des Ansatzes, gegensätzlich zur kombinierten C- und CPP-Analyse von *TypeChef*, auf dem Herausstellen von Präprozessorbedingungen für das Erreichen von Makrodefinitionen und Dateiinklusionen, um die dort vorhandene Hierarchie zu ermitteln [HMDL00].

Die Technik der symbolischen Ausführung der CPP wird in *TypeChef* nicht angewendet. Die Extraktion der CPP-Hierarchie kann jedoch als Grundlage für die Untersuchung der bedingt übersetzten Makrodefinitionen oder Inklusionen dienen.

7.2 Produktlinienorientierte Typsysteme

Ein produktlinienorientiertes Typsystem ist auf die Prüfung der Typsicherheit einer gesamten SPL ausgerichtet, ohne alle Variante schrittweise einzeln zu testen. Im Folgenden werden verschiedene Ansätze betrachtet, die eine SPL auf Typkorrektheit untersuchen.

FFJ_{PL}

In APEL et al. [AKGL09] wird Feature Featherweight Java Product Line (FFJPL) vorgestellt, die die Java-Teilmenge Featherweight Java (FJ) erweitert, um die merkmalsorientierte Programmierung zur Entwicklung einer SPL zu unterstützen. Für FFJPL wird ein Typsystem erschaffen, das die Typkorrektheit der gesamten SPL feststellt.

Die Produktlinie wird als wohlgetypt bezeichnet, wenn für alle generierbaren Varianten diese Eigenschaft nachgewiesen wird. Die Ausrichtung von *TypeChef* unterscheidet sich dahingehend, dass die Programmiersprache C und das Erkennen von fehlerhaften Referenzen im Fokus der Untersuchungen stehen.

CIDE/ CFJ

Das Werkzeug Colored Integrated Development Environment (CIDE) dient der Entwicklung von Produktlinien. CIDE verfolgt einen Annotationsansatz, der den merkmalspezifischen Quellcode durch das Einfärben des Hintergrundes kennzeichnet. In CIDE werden Typprüfungen für Java unter Berücksichtigung der vorhandenen Annotationen realisiert, die jedoch nicht vollständig sind [KAT⁺09].

Mittels Colored Featherweight Java (CFJ) wurde eine Erweiterung von FJ erschaffen, die die Annotationsform aus CIDE verwendet und zur Entwicklung von Softwareproduktlinien nutzbar ist [KA08b]. Das Typsystem von CFJ stellt sicher, dass alle Programmvarianten, die aus einer wohlgetypten SPL generiert werden können, wiederum wohlgetypte FJ-Programme sind. Das CFJ-Typsystem unterscheidet sich grundlegend in der verwendeten Zielsprache sowie der genutzten Annotationstechnik von dem Vorgehen in *TypeChef*.

safe composition

THAKER et al. [TBKC07] beschreiben das Werkzeug *safe composition* für AHEAD Produktlinien, das die Abwesenheit von Referenzen zu undefinierten Elementen, wie Klassen oder Methoden, garantiert. Der Ansatz vollzieht die Typprüfung einer SPL unter Verwendung eines SAT-Solvers. Dementsprechend wird untersucht, ob alle für die Implementierung aufgestellten Bedingungen durch das Merkmalmodell erfüllt werden. In *safe composition* wird, im Gegensatz zu *TypeChef*, die Typprüfung für einen Vertreter der Klasse von Kompositionsansätzen verfolgt. Weiterhin dient Referenzanalyse dem Auffinden von nicht definierten Elementen, wird jedoch nicht für die Programmiersprache C durchgeführt.

fmp2rsm

In CZARNECKI UND PIETROSK [CP06] wird das Eclipse-Plug-in *fmp2rsm* vorgestellt. Dieses Werkzeug dient dem Erstellen von Produktlinien für UML-Modelle und garantiert, dass alle Varianten eines merkmalsbasierten Modell-Templates wohlgeformt sind. Hierfür wird eine Möglichkeit bereitgestellt, die UML-Elemente mit einer Präsenzbedingung zu versehen und somit einem Merkmal zuzuordnen. Die Elemente, wie Klassen oder Verbindungen, werden bei der Generierung einer validen Konfiguration entfernt, wenn die Bedingung als falsch bewertet wird. Die Kontrolle der Wohlgeformtheit des Modell-Templates wird mit Hilfe eines SAT-Solvers realisiert. Die Präsenz und weitere Bedingungen, die die gewünschte Wohlgeformtheit festlegen, werden im Solver auf Fehler überprüft.

Grundlegend unterscheiden sich *fmp2rsm* und *TypeChef* in der Art der Variabilitäts erzeugung sowie des SPL-Anwendungsgebietes. Beide Systeme nutzen ein automatisiertes Lösungssystem, das die Korrektheit der Konfigurationen sicherstellt. Jedoch

erlaubt der CPP die Nutzung von komplexen Bedingungen, sodass ein SAT-Solver nicht ausreichend ist und ein CSP-Solver benötigt wird, um die Untersuchung der Präsenz zu realisieren.

KAPITEL 8

Zusammenfassung

Der C-Präprozessor ist ein gebräuchliches Werkzeug zur Erzeugung von Variabilität in Anwendungen. In einer Teilphase des Übersetzungsvorganges wird der Quellcode durch die Festlegung einer Merkmalauswahl transformiert und eine Variante des Programms erschaffen. Grundlegend werden hierbei der Funktionsumfang und die Programmstruktur hinsichtlich verschiedener Plattformen angepasst.

Die Entwicklung eines produktlinienorientierten Typsystems für CPP-konfigurierte Anwendungen stellt die Korrektheit aller konstruierbaren Varianten heraus, ohne jede Konfiguration einzeln auf Typfehler zu testen.

Im Rahmen der vorliegenden Arbeit wurde mit *TypeChef* ein Werkzeug prototypisch konzeptioniert und realisiert, das die enthaltene Variabilität mit Hilfe der statischen Referenzanalyse prüft. Ein Ziel dieser Arbeit war die Untersuchung der Realisierbarkeit eines solchen Werkzeuges. Dementsprechend standen die Identifikation von Grenzen und das Herausstellen von Problemen im Vordergrund der Arbeit.

Die Grundlage der Analyse bilden die Quellcodedokumente, die aus den verschränkten Konstrukten der Programmiersprache C sowie den Direktiven des CPP bestehen. Die Variabilität wird durch die Anweisungen der bedingten Übersetzung erzeugt. Mit dem partiellen C-Präprozessor wurde ein Mechanismus erschaffen, der den durchgesetzten Quellcode durch einen CPP verarbeiten lässt, jedoch ohne die Variabilität erzeugenden Anpassungen vorzunehmen. Diese Phase ist ein wichtiger Bestandteil der Untersuchung, da verschiedene komplexe Aufgaben im Zusammenhang mit dem CPP automatisiert und durch ein etabliertes Werkzeug ausgeführt werden.

Die expandierte Datei wird durch den erweiterten Parser in einen AST abgebildet. Der bestehende Parser für C wurde um einen Regelsatz für die fokussierten Direktiven ergänzt. Für die kombinierte Analyse von C und CPP wird die Stufe der Granularität dahingehend begrenzt, dass mindestens ein Statement vollständig umschlossen wird. Die Einhaltung der auf diesem Weg festgelegten Disziplin einer Annotation wird durch *TypeChef* sichergestellt. Eine undisziplinierte Ausführung ist mittels manueller Anpassungen in die korrekte Form zu überführen.

Aus dem AST eines zu untersuchenden Dokumentes werden alle für die statische Referenzanalyse notwendigen Informationen extrahiert. Die Programmiersprache C setzt sich aus verschiedenen Elementen zusammen, die mit Hilfe einer spezifischen Operation erstellt und über den eindeutigen Bezeichner referenzierbar ist. Die Definition/Deklaration sowie der Zugriff für ein Element werden anhand von Mustern im AST erkannt und in einer erweiterten Symboltabelle verwaltet. Jede zugreifende Operation löst die Bildung eines Referenzpaares mit dem erschaffenden Eintrag in der Tabelle

aus. Das Paar beinhaltet Präsenzinformationen, die die annotierende CPP-Hierarchie der beiden Bestandteile ausdrücken.

Im letzten Schritt von *TypeChef* werden die Bedingungen für den Verbleib der Elemente extrahiert und die Gesamtbedingung für einen Typfehler formuliert. Die Aussagen in den Direktiven erlauben komplexe Makrogleichungen. Dementsprechend wird insgesamt ein CSP definiert, das in dem Lösungssystem Choco überprüft wird. Eine Komponente in *TypeChef* transformiert die jeweiligen Hierarchien und testet die Aussage über einen Typfehler auf eine erfüllende Makrobelegung. Ein Paar ist genau dann fehlerfrei, wenn der Solver zu keiner Lösung gelangt. Die Resultate für alle untersuchten Referenzpaare fassen die Anzahl an durchgeführten Test und die dabei herausgestellten Typfehler zusammen. Darauf aufbauend können die betroffenen Quellcodebereiche korrigiert werden.

Die Funktionsweise von *TypeChef* wurde anhand bestehender C-Anwendungen untersucht und die Ergebnisse bewertet. Der Webserver boa wurde vollständig analysiert. Aus der Menge der fehlerhaft gekennzeichneten Paare wurden verschiedene Situationen ausgewählt und durch die parametrisierte Übersetzung der Anwendung nachgestellt. Der Vergleich der Ergebnisse aus *TypeChef* und der manuellen Kontrolle bestätigen die Existenz von Inkorrektheiten in der Konfigurationsstruktur. Für zwei einzelne Merkmale und eine Merkmalkombination konnte die Generierung von Varianten, die der Compiler als fehlerhaft ablehnt, nachgewiesen werden. *TypeChef* hat demnach drei Fehler in der Anwendung boa gefunden, die die Menge der korrekten Konfigurationen verkleinert. Für alle nicht nachweisbaren Fehler konnte in einer gesonderten Analyse der Quellcodedokumente die Herkunft dieser Fehlanalysen identifiziert werden.

In dem Projekt gretl werden CPP-Direktiven angewendet, die über die Abfrage der Definition eines Makros hinausgehen. Die Nutzung der Direktiven wurde anhand einer enthaltenen Datei untersucht. Um die in *TypeChef* unterstützten CPP-Anweisungen zu zeigen, wurden die vorhandenen Bedingungen modifiziert und erweitert. Die dabei entstehenden Annotationen demonstrieren, dass auch komplexe Präsenzkonditionen verarbeitet werden können.

Das Ziel der Arbeit war die Konzeptionierung und Umsetzung eines Werkzeuges für die Analyse von C-Anwendungen, in denen die Variabilität mit Hilfe des CPP erzeugt wird. *TypeChef* ist die prototypische Realisierung dieser Anforderungen und ermöglicht die statische Referenzprüfung für alle aus einer Anwendung generierbaren Varianten.

Ausblick

Während der Entwicklung von *TypeChef* sowie der Durchführung der Fallstudien wurden das Ziel verfolgt, Grenzen und Probleme für einzelne Phasen aufzudecken. Die anschließend aufgeführten Punkte fassen die Perspektiven aus Kapitel 6 für eine Erweiterung von *TypeChef* zusammen:

- Die Beeinflussung der Quellcodetransformation durch die variable Gestaltung der CPP-Anweisungen selbst wird seitens des partiellen Präprozessors nicht korrekt aufgelöst. Die Entwicklung eines spezialisierten, produktlinienorientierten

Makroprozessors erlaubt die Integration von eigenen, konfigurationsbezogenen Expansions- und Inklusionsmechanismen.

- Die Verarbeitung von plattformspezifischen Anpassungen muss detailliert betrachtet werden, da *TypeChef* hierbei nur einfache Blöcke unterstützt. Eine Erweiterung auf komplexe Zusammensetzungen sowie Kombinationen aus Systemmakros und Merkmalmakros vereinfacht den Umgang mit nicht analysierbaren Plattformkonfigurationen.
- Sobald bei der Zusammenstellung eines Referenzpaares mehrere erstellende Operationen zugeordnet werden, ist ein zusätzlicher Test zu integrieren, der sicherstellt, dass die Alternativen nicht gleichzeitig existieren.
- Der CSP-Transformator ist zur Unterstützung des vollen Funktionsumfanges der CPP-Ausdrücke zu ergänzen. Folglich können zusätzlich die fehlenden Operationen sowie Operanden verarbeitet werden.
- Die Disziplinverletzung durch eine Annotation wird zwar festgestellt, muss jedoch manuell restrukturiert werden. Der automatisierte Ablauf der Disziplinierung vergrößert das Spektrum der untersuchbaren Anwendungen, speziell, wenn eine hohe Anzahl an undisziplinierten Markierungen herausgestellt wird.
- Durch die Erweiterung der Referenzanalyse um Typinformationen können zusätzlich Prüfungen realisiert werden, die neben dem Vorhandensein von referenzierten Elementen die Typkorrektheit in der jeweiligen Verwendung analysieren.
- In weiteren und umfangreichen Fallstudien ist die Anwendbarkeit von *TypeChef* zu überprüfen. Intern werden eine GNU-C-Grammatik und der GCC als partieller C-Präprozessor verwendet. Folglich wird das Spektrum auf Anwendungen festgelegt, die diesem oder einem darin enthaltenen Sprachstandard für C folgen. Nach der Erweiterung von *TypeChef* hinsichtlich der aufgeführten Problemfelder müssen weitere Untersuchungen durchgeführt werden, um das Anwendungsspektrum weiter zu vergrößern und die Ergebnisse der Analysen zu verbessern.

Das im Rahmen dieser Arbeit entwickelte Werkzeug ist ein erster Schritt zu einem produktlinienorientierten Typsystem für die Programmiersprache C und die variable Gestaltung durch den CPP.

Literaturverzeichnis

- [ADMTH09] ADAMS, B. ; DE MEUTER, W. ; TROMP, H. ; HASSAN, A.E.: Can we refactor conditional compilation into aspects? In: *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD)*, ACM, 2009, S. 243–254
- [ADPB02] AVERSANO, L. ; DI PENTA, M. ; BAXTER, I.D.: Handling preprocessor-conditioned declarations. In: *Proceedings of the 2nd International Workshop on Source Code Analysis and Manipulation (SCAM)*, IEEE Computer Society, 2002, S. 83–92
- [AKGL09] APEL, S. ; KÄSTNER, C. ; GRÖSSLINGER, A. ; LENGAUER, C.: Type-Safe Feature-Oriented Product Lines / Department of Informatics and Mathematics, University of Passau. 2009 (MIP-0909). – Forschungsbericht
- [AKL09] APEL, S. ; KÄSTNER, C. ; LENGAUER, C.: Vergleich und Integration von Komposition und Annotation zur Implementierung von Produktlinien. In: *Software Engineering 2009 – Fachtagung des GI-Fachbereichs Softwaretechnik* Bd. 143, Gesellschaft für Informatik (GI), 2009 (Lecture Notes in Informatics), S. 101–112
- [Ape07] APEL, S.: *The Role of Features and Aspects in Software Development*, University of Magdeburg, Germany, Dissertation, 2007
- [Bat03] BATORY, D.: A Tutorial on Feature Oriented Programming and Product-Lines. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2003, S. 753–754
- [Bat05] BATORY, D.: Feature Models, Grammars, and Propositional Formulas. In: *Proceedings of the 9th International Software Product Lines Conference (SPLC)* Bd. 3714, Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 7–20
- [BBC⁺10] BESSEY, A. ; BLOCK, K. ; CHELF, B. ; CHOU, A. ; FULTON, B. ; HALLEM, S. ; HENRI-GROS, C. ; KAMSKY, A. ; MCPPEAK, S. ; ENGLER, D.: A few billion lines of code later: using static analysis to find bugs in the real world. In: *Communications of the ACM* 53 (2010), Nr. 2, S. 66–75
- [BCDVM02] BRYANT, A. ; CATTON, A. ; DE VOLDER, K. ; MURPHY, G. C.: Explicit programming. In: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, ACM, 2002, S. 10–18

- [BCS00] BATORY, D. ; CARDONE, R. ; SMARAGDAKIS, Y.: Object-Oriented Frameworks and Product-Lines. In: *Proceedings of the 1st Software Product Lines Conference : Experience and Research Directions (SPLC)*, 2000, S. 227
- [BM01] BAXTER, I.D. ; MEHLICH, M.: Preprocessor conditional removal by simple partial evaluation. In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, IEEE Computer Society, 2001, S. 281–290
- [BMAC05] BENAVIDES, D. ; MARTÍN-ARROYO, P.T ; CORTÉS, A.R.: Automated Reasoning on Feature Models. In: *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE)* Bd. 3520, Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 491–503
- [BP08] BOVET, J. ; PARR, T.: ANTLRWorks: an ANTLR grammar development environment. In: *Software: Practice and Experience* 38 (2008), Nr. 12, S. 1305–1332
- [Bre09] BRESLAV, A.: Creating modular and reusable DSL textual syntax definitions with Grammatic/ANTLR. In: *Computing Research Repository* arXiv:0902.2621 (2009), Nr. 1
- [BSMAC06] BENAVIDES, D. ; SEGURA, S. ; MARTÍN-ARROYO, P.T. ; CORTÉS, A.R.: Using Java CSP Solvers in the automated Analyses of Feature Models. In: *Proceedings of the 1st International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)* Bd. 4143, Springer-Verlag, 2006 (Lecture Notes in Computer Science), S. 399–408
- [BSRC10] BENAVIDES, D. ; SEGURA, S. ; RUIZ-CORTÉS, A.: Automated Analysis of Feature Models 20 years Later: a Literature Review. In: *Information Systems* (2010)
- [BSTC07] BENAVIDES, D. ; SEGURA, S. ; TRINIDAD, P. ; CORTÉS, A.R.: FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In: *Proceedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)* Bd. 2007-01, 2007 (Lero Technical Report), S. 129–134
- [Car97] CARDELLI, L.: Type Systems. In: *The Computer Science and Engineering Handbook*. CRC Press, 1997, S. 2208–2236
- [CE99] CZARNECKI, K. ; EISENECKER, U. W.: Components and Generative Programming. In: *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT international Symposium on Foundations of Software Engineering (ESEC/FSE)* Bd. 1687, Springer-Verlag, 1999 (Lecture Notes in Computer Science), S. 2–19

- [CE00] CZARNECKI, K. ; EISENECKER, U. W.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000
- [CN01] CLEMENTS, P. ; NORTHROP, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001
- [CP06] CZARNECKI, K. ; PIETROSZEK, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, ACM, 2006, S. 211–220
- [CPR07] COPPIT, D. ; PAINTER, R.R. ; REVELLE, M.: Spotlight: A prototype tool for software plans. In: *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2007, S. 754–757
- [CR99] CHANDRA, S. ; REPS, T.W.: Physical Type Checking for C. In: *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, ACM, 1999, S. 66–75
- [CR05] CIRIC, M. ; RANCIC, S.R.: Parsing in different languages. In: *Facta Universitatis (Niš), Ser. Elec. Energ* 18 (2005), Nr. 2, S. 299–307
- [CT04] COOPER, K.D. ; TORCZON, L.: *Engineering a compiler*. Elsevier, 2004
- [CW07] CZARNECKI, K. ; WASOWSKI, A.: Feature Diagrams and Logics: There and Back Again. In: *Proceedings of the 11th International Conference on Software Product Lines (SPLC)*, IEEE Computer Society, 2007, S. 23–34
- [Cza98] CZARNECKI, K.: *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, Technical University of Ilmenau, Germany, Dissertation, 1998
- [Cza05] CZARNECKI, K.: Overview of Generative Software Development. In: *Proceedings of the International Workshop on Unconventional Programming Paradigms (UPP)* Bd. 3566, Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 326–341
- [Dij76] DIJKSTRA, E. W.: *A Discipline of Programming*. Prentice Hall, 1976
- [Fav95] FAVRE, J.M.: The CPP paradox. In: *Proceedings of the 9th European Workshop on Software Maintenance*, 1995
- [Fav96] FAVRE, J.M.: Preprocessors from an Abstract Point of View. In: *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 1996, S. 329–

- [Fav97] FAVRE, J.M.: Understanding-in-the-large. In: *Proceedings of the 5th International Workshop on Program Comprehension (IWPC)*, IEEE Computer Society, 1997, S. 29–38
- [FDK93] FORBUS, K.D. ; DE KLEER, J.: *Building Problem Solvers*. The MIT Press, 1993
- [FKW04] FRANCYK, B. ; KIEBUSCH, S. ; WERNER, A.: Domänenanalyse (Stakeholder) und Qualitätsmetriken für Softwareproduktlinien / Universität Leipzig. 2004 (2). – Forschungsbericht
- [GJ03] GARRIDO, A. ; JOHNSON, R.E.: Refactoring C with Conditional Compilation. In: *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, 2003, S. 323–326
- [GJ05] GARRIDO, A. ; JOHNSON, R.E.: Analyzing Multiple Configurations of a C Program. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 2005, S. 379–388
- [HMDL00] HU, Y. ; MERLO, E. ; DAGENAIS, M. ; LAGUË, B.: C/C++ Conditional Compilation Analysis using Symbolic Execution. In: *Proceedings of the 16th International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 2000, S. 196–206
- [HO93] HARRISON, W. ; OSSHER, H.: Subject-Oriented Programming (A Critique of Pure Objects). In: *Proceedings of the 8th annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* Bd. 28, ACM, 1993, S. 411–428
- [Int99] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO/IEC 9899-1999: Programming Languages-C*. 1999
- [JBZZ03] JARZABEK, S. ; BASSETT, P. ; ZHANG, H. ; ZHANG, W.: XVCL: XML-based Variant Configuration Language. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2003, S. 810–811
- [JF88] JOHNSON, R.E. ; FOOTE, B.: Designing reuseable Classes. In: *Journal of Object-Oriented Programming* (1988)
- [JPC⁺08] JUSSIEN, N. ; PRUD'HOMME, C. ; CAMBAZARD, H. ; ROCHART, G. ; LABURTHE, F.: choco: an Open Source Java Constraint Programming Library. In: *Proceedings of the 3rd International CSP Solver Competition held in conjunction with the 13th International Conference on Principles and Practice of Constraint Programming (CPAI/CP)*, 2008, S. 7–14

- [KA08a] KÄSTNER, C. ; APEL, S.: Integrating Compositional and Annotative Approaches for Product Line Engineering. In: *Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLe)*, Department of Informatics and Mathematics, University of Passau, 2008, S. 35–40
- [KA08b] KÄSTNER, C. ; APEL, S.: Type-Checking Software Product Lines - A Formal Approach. In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, 2008, S. 258–267
- [KA09] KÄSTNER, C. ; APEL, S.: Virtual Separation of Concerns - A Second Chance for Preprocessors. In: *Journal of Object Technology* 8 (2009), Nr. 6, S. 59–78
- [KAK08] KÄSTNER, C. ; APEL, S. ; KUHLEMANN, M.: Granularity in Software Product Lines. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, ACM, 2008, S. 311–320
- [KAK09] KÄSTNER, C. ; APEL, S. ; KUHLEMANN, M.: A model of refactoring physically and virtually separated features. In: *Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE)*, ACM, 2009, S. 157–166
- [KAS10] KÄSTNER, C. ; APEL, S. ; SAAKE, G.: Virtuelle Trennung von Belangen (Präprozessor 2.0). In: *Software Engineering 2010 - Fachtagung des GI-Fachbereichs Softwaretechnik* Bd. 159, Gesellschaft für Informatik (GI), 2010 (Lecture Notes in Informatics), S. 165–176
- [KAT⁺09] KÄSTNER, C. ; APEL, S. ; TRUJILLO, S. ; KUHLEMANN, M. ; BATORY, D.: Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In: *Proceedings of the 47th International Conference on Objects, Components, Models and Patterns (TOOLS EUROPE)* Bd. 33, Springer-Verlag, 2009 (Lecture Notes in Business Information Processing), S. 175–194
- [KCH⁺90] KANG, K. C. ; COHEN, S. G. ; HESS, J. A. ; NOVAK, W. E. ; PETERSON, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Carnegie-Mellon University Software Engineering Institute. 1990 (CMU/SEI-90-TR-21). – Forschungsbericht
- [Kin76] KING, J.C.: Symbolic execution and program testing. In: *Communications of the ACM* 19 (1976), Nr. 7, S. 385–394
- [KLD02] KANG, K. C. ; LEE, J. ; DONOHOE, P.: Feature-Oriented Product Line Engineering. In: *IEEE Software* 19 (2002), Nr. 4, S. 58–65
- [KR88] KERNIGHAN, B.W. ; RITCHIE, D.M.: *The C programming language*. Prentice Hall, Inc., 1988

- [KS94] KRONE, M. ; SNETLING, G.: On the Inference of Configuration Structures from Source Code. In: *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 1994, S. 49–57
- [Käs10] KÄSTNER, C.: *Virtual Separation of Concerns: Preprocessors 2.0*, University of Magdeburg, Germany, Dissertation, 2010
- [KTA08] KÄSTNER, C. ; TRUJILLO, S. ; APEL, S.: Visualizing Software Product Line Variabilities in Source Code. In: *Proceedings of the 2nd International SPLC Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, 2008, S. 303–313
- [LKL⁺97] LOPES, C.V. ; KICZALES, G. ; LAMPING, J. ; MENDHEKAR, A. ; MAEDA, C. ; LOPES, C. ; LOINGTIER, J. ; IRWIN, J.: Aspect-Oriented Programming. In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag, 1997, S. 483–496
- [LR09] LAHRENS, B. ; RAYMAN, G.: *Objektorientierte Programmierung*. Galileo Computing, 2009
- [LZW⁺08] LIU, S. ; ZHANG, R. ; WANG, D. ; SUN, H. ; CHEN, Y. ; LI, L.: Implementing of Gaussian Syntax-Analyzer Using ANTLR. In: *Proceedings of the 7th International Conference on Cyberworlds*, IEEE Computer Society, 2008, S. 613–618
- [MB05] MCCLOSKEY, B. ; BREWER, E.A.: ASTEC: a new approach to refactoring C. In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE)*, ACM, 2005, S. 21–30
- [MP03] MUTHIG, D. ; PATZKE, T.: Generic Implementation of Product Line Components. In: *Proceedings of the International Conference NetObject-Days on Objects, Components, Architectures, Services, and Applications for a Networked World (NODE), Revised Papers* Bd. 2591, Springer-Verlag, 2003 (Lecture Notes in Computer Science), S. 313–329
- [OJ97] O’CALLAHAN, R. ; JACKSON, D.: Lackwit: A Program Understanding Tool Based on Type Inference. In: *Proceedings of the 19th International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 1997, S. 338–348
- [OT01] OSSHER, Harold ; TARR, Peri: Using multidimensional separation of concerns to (re)shape evolving software. In: *Communications of the ACM* 44 (2001), Nr. 10, S. 43–50

- [Pad09] PADIOLEAU, Y.: Parsing C/C++ Code without Pre-processing. In: *Proceedings of the 18th International Conference on Compiler Construction (CC)* Bd. 5501, Springer-Verlag, 2009 (Lecture Notes in Computer Science), S. 109–125
- [Par76] PARNAS, D. L.: On the Design and Development of Program Families. In: *Software Engineering, IEEE Transactions on SE-2* (1976), Nr. 1, S. 1–9
- [Par79] PARNAS, D. L.: Designing Software for Ease of Extension and Contraction. In: *Software Engineering, IEEE Transactions on* 5 (1979), Nr. 2, S. 128–138
- [Par07] PARR, T.: *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007
- [Par08] PARR, T.: The Reuse of Grammars with Embedded Semantic Actions. In: *Proceedings of the 16th International Conference on Program Comprehension (ICPC)*, IEEE Computer Society, 2008, S. 5–10
- [PBL05] POHL, K. ; BÖCKLE, G. ; LINDEN, Frank J. van d.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005
- [Pie02] PIERCE, B.C.: *Types and programming languages*. The MIT Press, 2002
- [Pie03] PIERCE, B.C.: Types and Programming Languages: The Next Generation. In: *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society, 2003, S. 32
- [PQ95] PARR, Terence J. ; QUONG, Russell W.: ANTLR: A Predicated- $LL(k)$ Parser Generator. In: *Software: Practice and Experience* 25 (1995), Nr. 7, S. 789–810
- [Pri93] PRINZ, P.: *ANSI-C-Guide: das kompakte Nachschlagewerk für alle C-Compiler*. IWT Verlag, 1993
- [SC92] SPENCER, H. ; COLLYER, G.: #ifdef Considered Harmful, or Portability Experience with C News. In: *Proceedings of the Summer USENIX Technical Conference*, USENIX Association, 1992, S. 185–197
- [Str02] STROUSTRUP, B.: C and C++: Siblings. In: *The C/C++ Users Journal* 20 (2002), Nr. 7, S. 28–36
- [SW10] STALLMAN, R.M. ; WEINBERG, Z. ; FREE SOFTWARE FOUNDATION, INC (Hrsg.): *The C Preprocessor*. fourth edition. Boston, MA, USA: Free Software Foundation, Inc, 2010
- [Szy97] SZYPERSKI, C.: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1997

- [TBKC07] THAKER, S. ; BATORY, D. ; KITCHIN, D. ; COOK, W.R.: Safe composition of product lines. In: *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE)*, ACM, 2007, S. 95–104
- [VB03] VIDÁCS, L. ; BESZÉDES, A.: Opening Up The C/C++ Preprocessor Black Box. In: *Proceedings of the 8th Symposium on Programming Languages and Software Tools (SPLST)*, University of Kuopio, Department of Computer Science, 2003, S. 45–57
- [Vit03] VITTEK, M.: Refactoring Browser with Preprocessor. In: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society, 2003, S. 101–110
- [Wit96] WITHEY, J.: Investment Analysis of Software Assets for Product Lines / Software Engineering Institute, Carnegie Mellon University. 1996 (CMU/SEI-96-TR-010). – Forschungsbericht
- [Wol09] WOLF, J.: *C von A bis Z*. Galileo Computing, 2009
- [WSB⁺08] WHITE, J. ; SCHMIDT, D.C. ; BENAVIDES, D. ; TRINIDAD, P. ; CORTÉS, A.R.: Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In: *Proceedings of the 12th International Conference on Software Product Lines (SPLC)*, IEEE Computer Society, 2008, S. 225–234

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit in allen Teilen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel genutzt habe. Alle wörtlich oder sinngemäß übernommenen Textstellen habe ich als solche kenntlich gemacht.

Magdeburg, den 30. August 2010

Andy Kenner

