Steffen Hölldobler et. al. (Hrsg.): Ausgezeichnete Informatikdissertationen 2015,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 189

Analyse und Transformation konfigurierbarer Systeme[1]

Jörg Liebig[2]

Abstract: Viele Softwaresysteme, wie z.B. das Betriebssystem Linux, stellen zum Teil tausende von Konfigurationsoptionen bereit, mit deren Hilfe eine zum Teil astronomisch große Anzahl unterschiedlicher Varianten (für Linux vom Smartphone bis zum Supercomputer) erstellt werden kann. Die Variantenvielfalt dieser Systeme stellt Softwareentwickler vor enorme Herausforderungen hinsichtlich ihrer Korrektheit, da entsprechende Entwicklungswerkzeuge fehlen. In dieser Arbeit beschreiben wir skalierbare, variabilitätsgewahre Techniken für die Entwicklung von Analyse- und Transformationswerkzeugen, die mit der Variantenvielfalt konfigurierbarer Systeme umgehen können. Wir zeigen, dass diese Techniken bestehenden Verfahren hinsichtlich Laufzeit, Aufwand zur Anwendung und Vollständigkeit überlegen sind.

1 Einleitung

Statische Analyse- und Transformationswerkzeuge für Quellcode gehören zur Standardausrüstung eines Softwareentwicklers. Ihre Anwendung vereinfacht die tägliche Arbeit bezüglich der Pflege und Weiterentwicklung des Quellcodes entscheidend. Daher beeinflussen sie maßgeblich die Effizienz und Produktivität von Entwicklern. Ferner hat die Anwendung der Werkzeuge auch finanzielle Vorteile, denn Programmierfehler können so bereits frühzeitig im Software-Entwicklungsprozess aufgespürt bzw. vermieden werden [KM03]. Das hat zur Folge, dass sich Wartungs- und Entwicklungskosten entscheidend reduzieren lassen.

In der Praxis werden Softwaresysteme häufig aufgrund unterschiedlicher Anforderungen als konfigurierbare Systeme entwickelt. Für deren Entwicklung greifen Entwickler häufig auf `#ifdef` Direktiven des Präprozessors CPP zurück, um Variabilität zur Compilezeit eines Systems umzusetzen. Variable Codefragmente werden mit Hilfe von `#ifdef` Direktiven annotiert und über Konfigurationsoptionen kontrolliert. Deren Auswahl/Abwahl im Konfigurationsprozess eines Systems entscheidet über den Funktionsumfang der Systemvarianten. Die Nutzung von Annotationen für die Entwicklung konfigurierbarer Systeme ist einfach zu lernen und anzuwenden, sodass sie von vielen Entwicklern für Konfigurationszwecke verwendet werden. Existierende konfigurierbare Systeme, wie zum Beispiel der LINUX Kernel, stellen für Konfigurationszwecke häufig tausende von Konfigurationsoptionen bereit, mit deren Hilfe Billiarden unterschiedlicher Systemvarianten (von eingebetteten Systemen über Smartphones und PCs, bis hin zu Supercomputern) auf Abruf erstellt werden können.

Obwohl konfigurierbare Systeme und entsprechende Implementierungsansätze in der Praxis weit verbreitet sind, fehlen passende Entwicklungswerkzeuge. Bestehende Analyse- und Transformationswerkzeuge sind nicht für die Entwicklung konfigurierbarer Systeme vorbereitet, d.h. sie können nicht mit Variabilität im Quellcode umgehen. Das hat

[1] Analysis and Transformation of Configurable Systems [Li15a]

[2] Method Park, joerg.liebig@methodpark.com

zur Folge, dass bestehende Werkzeuge fehlerhafte oder unvollständige Ergebnisse liefern. Gleichzeitig lassen sich immer komplexere Softwaresysteme häufig ohne entsprechende Werkzeugunterstützung nicht mehr effizient entwickeln. Trotz ihrer weiten Verbreitung werden Implementierungsansätze auf der Basis von Präprozessor-Direktiven in der Forschung häufig kritisiert. Die Kritik richtet sich unter anderem auch an die fehlenden Entwicklungswerkzeuge (z.B. [SC92, Fa96]). Obwohl die vorgebrachte Kritik zum Teil seit mehr als 20 Jahren besteht, gibt es keine Anzeichen für grundlegende Verbesserungen.

Der Beitrag dieser Arbeit liegt im Bereich der Entwicklung von variabilitätsgewahren Techniken zur Analyse und Transformation von konfigurierbaren Systemen [Li15a]. Diese Techniken nutzen Gemeinsamkeiten zwischen verschiedenen Systemvarianten aus, um den Aufwand für Analysen und Transformationen entscheidend zu reduzieren. Die Durchführung von Analyse und Transformationen für alle Systemvarianten mit Hilfe von brute-force (jede Variante wird einzeln betrachtet) ist für praktische Systeme mit einer häufig astronomisch großen Anzahl an Systemvarianten nicht möglich. Die dafür notwendigen Ressourcen (Rechenleistung und Speicherplatzverbrauch) übertreffen verfügbare Ressourcen um ein Vielfaches. In dieser Arbeit entwickeln wir neue Analyseansätze für die statische Analyse von Billiarden unterschiedlicher Systemvarianten und vergleichen sie mit traditionellen Verfahren auf der Basis von Sampling, also der individuellen Betrachtung einiger weniger Systemvarianten mit Hilfe von Analysen, die nicht variabilitätsgewahr sind. Der Vergleich zeigt, dass variabilitätsgewahre Analysen im Hinblick auf Vollständigkeit (alle gültigen Systemvarianten werden betrachtet), Effizienz (sie haben eine geringere Laufzeit) und Skalierbarkeit (Anwendbarkeit auch für große Softwaresysteme) Sampling-Verfahren überlegen sind. Wir zeigen die praktische Anwendbarkeit variabilitätsgewahre Analysen auch für Softwaresysteme der Größe des Betriebssystems LINUX.

Auf der Grundlage variabilitätsgewahrer Analysen entwickeln wir ein Transformationswerkzeug für C namens MORPHEUS, das drei Standard-Refactorings umsetzt (RENAME IDENTIFIER, EXTRACT FUNCTION und INLINE FUNCTION). MORPHEUS berücksichtigt Variabilität im Quellcode direkt und räumt mit bestehenden Defiziten (fehlende Vollständigkeit, Anwendung von Heuristiken und mangelnde Skalierbarkeit) existierender Transformationswerkzeuge auf. Gleichzeitig werden alle Systemvarianten eines konfigurierbaren Systems berücksichtigt, damit das Systemverhalten aller Varianten vor und nach der Transformation gleich bleibt. Um dies sicherzustellen erweitern wir einen gängigen Testansatz für Transformationswerkzeuge um die Fähigkeit zur Überprüfung von Variabilität. Wir zeigen damit anhand praktischen Fallstudien, dass variabilitätsgewahre Transformationen effektiv sind und auch für größere Softwaresysteme skalieren.

2 Zum Verständnis von Präprozessor Annotationen

Obwohl Präprozessor-basierte Implementierungsansätze in der Praxis seit mehr als 40 Jahren eingesetzt werden, ist recht wenig über ihren Einsatz zur Implementierung konfigurierbarer Systeme bekannt. Für konfigurierbare Systeme fehlen häufig bereits einfache Informationen, wie beispielsweise die Anzahl der Konfigurationsoptionen. Um eine umfassende Übersicht zur Verwendung von `#ifdef` Annotationen in der Praxis zu erhalten, haben wir ihre Verwendung im Detail untersucht [Li10, Li11]. Grundlage für beide Studien war die

Goal Question Metric (GQM) Methode zur systematischen Erfassung und Messung des Einflusses von **#ifdef** Annotationen auf die Entwicklung von Werkzeugen. Im Einzelnen haben wir die Beziehungen von Konfigurationsoptionen untereinander, die Implementierung variablen Quellcodes mit Hilfe von **#ifdef**s und die Disziplin von **#ifdef** Annotationen untersucht. Für deren qualitative Erfassung wurde eine Reihe unterschiedlicher Metriken definiert. Zwei Beispiele dieser Metriken sind die Anzahl der Konfigurationsoptionen und die Verschachtelungstiefe von **#ifdef** Annotationen im Quellcode. Beide Metriken erlauben eine grobe Abschätzung der möglicherweise exponentiellen Komplexität konfigurierbarer Systeme und sie bestimmen maßgeblich ihren Analyse- und Transformationsaufwand.

Für 42 open-source Softwaresysteme aus unterschiedlichen Bereichen (z.B. Datenbanken, Web-Server oder Anwendungsprogramme) und verschiedener Größe haben wir die Metriken erhoben. Obwohl einige Softwaresysteme eine große Anzahl von Konfigurationsoptionen haben, konnten wir keine exponentielle Komplexität durch Verschachtelung im Quellcode finden. Dennoch haben wir eine Reihe wichtiger Anforderungen hinsichtlich der Entwicklung von Werkzeugen bestimmt. Zwei Beispiele sind: 1) **#ifdef**s bestehen häufig aus komplexen, Booleschen Ausdrücken von Konfigurationsoptionen und 2) Variabilität im Quellcode tritt in unterschiedlichen Bereichen auf (von annotierten Funktionen bis hin zu einzelnen Klammern). Die Berücksichtigung dieser und weiterer Anforderungen ist für die Entwicklung von Entwicklungswerkzeugen unerlässlich.

Darüber hinaus haben wir eine wichtige Eigenschaft von Präprozessor-basierten Implementierungsansätzen im Quellcode untersucht: Disziplin von **#ifdef** Annotationen [Li11]. Die Disziplin hat einen wesentlichen Einfluss auf die Erzeugung von Abstraktionen, die in Analyse- und Transformationswerkzeugen zum Einsatz kommen (Abschnitt 3). Das Problem ist, dass **#ifdef** Annotationen nicht an die syntaktische Struktur von Quellcode gebunden sind. D.h. Entwickler können **#ifdef**s für Annotationen beliebiger Codefragmente (z.B. Funktionen oder Anweisungen) bis hin zu einzelnen Zeichen, wie z.B. eine Klammer, verwenden. Gerade letztere Art von Annotationen, die wir undisziplinierte Annotationen nennen, können zu Syntaxfehlern im Quellcode führen [Kä09]. Ferner erlauben sie keine 1:1 Abbildung in variabilitätsgewahren Abstraktionen die für effiziente Entwicklungswerkzeuge, wie wir sie vorstellen, notwendig sind. Mit unserer Definition von disziplinierten und undisziplinierten Annotationen [Li11] haben wir alle Annotationen in den 42 Softwaresystemen untersucht. Dabei zeigte sich, dass bereits 85 % aller Annotationen diszipliniert sind. Allerdings war nur ein einziges von allen Softwaresystemen frei von undisziplinierten Annotationen, woraus folgt dass Entwicklungswerkzeuge mit diesen umgehen können müssen.

3 Die Analyse von C Code im Kontext von #ifdef Annotationen

Mit Sampling-Verfahren und variabilitätsgewahren Analysen existieren zwei Analyseansätze für konfigurierbarer Systeme [Th14]. Sampling basiert auf der Idee, dass die Analyse einer repräsentativen Menge ausreichende Ergebnisse für die Programmanalyse liefert. Mit Hilfe einer Heuristik wird eine Teilmenge aller gültigen Systemvarianten ausgewählt und mit traditionellen, nicht-variabilitätsgewahren Analysewerkzeugen untersucht. Dabei werden Gemeinsamkeiten, die von unterschiedlichen Systemvarianten geteilt werden, mehrfach analysiert (Redundanz), sodass Analyseergebnisse für ähnliche Varianten mehrfach

berechnet werden. Obwohl die Ergebnisse Sampling-basierter Verfahren notwendigerweise unvollständig sind, ist Sampling Standard in der Analyse konfigurierbarer Systeme [Th14].

Im Gegensatz zu Sampling arbeiten variabilitätsgewahre Analysen direkt auf der variablen Quellcodebasis. Alle durch `#ifdef` Direktiven auftretenden lokalen Variationen im Quelltext werden während der Analyse berücksichtigt, sodass ihre Ergebnisse alle Varianten einschließen (Vollständigkeit). Trotzdem werden Gemeinsamkeiten und Unterschiede zwischen verschiedenen Systemvarianten nur einmal analysiert, wodurch der Analyseaufwand gemessen an allen Varianten erheblich sinkt. Obwohl Sampling und variabilitätsgewahre Analysen bereits seit einiger Zeit in der Wissenschaft bekannt sind, existiert keine eingehende Untersuchung der beiden Ansätze hinsichtlich ihrer Anwendung in der Praxis. Zur Schließung dieser Lücke untersuchen wir beide Ansätze hinsichtlich ihrer Laufzeit, des Aufwands für ihren Einsatz und der Fehlererfassung. Alle drei Kriterien sind wichtige Voraussetzungen für den Einsatz statischer Analysen in der Praxis.

Laufzeit. Analysewerkzeuge arbeiten typischerweise auf Abstraktionen (wie beispielsweise Abstract Syntax Tree (AST) und Control-flow Graph (CFG)) des Quellcodes, die alle notwendigen Informationen für die Programmanalyse bereitstellen. Damit konfigurierbare Systeme effizient analysiert werden können, müssen die verwendeten Abstraktionen mit Variabilitätsinformationen angereichert werden [Wa14]. Dies erlaubt, dass Gemeinsamkeiten und Unterschiede eines konfigurierbaren Systems nur einmal analysiert werden müssen. Für die Erstellung von ASTs nutzen wir das Parser-Framework TYPECHEF. TYPECHEF erstellt 1:1 Repräsentationen von C Code mit `#ifdef` Annotationen und behandelt undisziplinierte Annotationen automatisch [Kä11]. Variable Codefragmente (`#ifdef`s) werden durch `Choice` Knoten repräsentiert und stellen somit eine einheitliche Repräsentation aller Systemvarianten dar (Abbildung 1).

Ausgehend von variablen ASTs haben wir einen variablen CFG entwickelt, der alle Programmausführungspfade (d.h. die Nachfolgerbeziehung von Programmanweisungen) im Softwaresystem repräsentiert [Li13]. In Abbildung 3 rechts unten wird einen Ausschnitt des variablen CFGs aus Abbildung 1 gezeigt. Die Zuweisung `c += b;` in Zeile 13 ist optional und wird im CFG durch Annotationen an den Kanten zwischen benachbarten Anweisungen repräsentiert (Zeile 12 bis 15). Auf der Grundlage von CFGs lassen sich leichtgewichtige Datenfluss-Berechnungen definieren, die definierte Programmeigenschaften (beispielsweise Programmierfehler) durch statische Analyse von Programmen bestimmen. Eine klassische Datenfluss-Analyse ist die Liveness-Berechnung. Sie bestimmt alle Variablen in einer Funktion, die vor dem nächsten Schreiben gelesen werden, und kann für die Analyse unbenutzten Quellcodes verwendet werden. Mit Hilfe unserer CFGs lässt sich die Liveness-Berechnung variabilitätsgewahr machen.

Variabilitätsgewahre Analysen erzeugen einen Mehraufwand durch die Überprüfung von Konfigurationsoptionen und deren Beziehungen untereinander. Fragestellungen, wie zum Beispiel ob eine Systemvariante gültig ist oder ob zwei Konfigurationsoptionen zusammen ausgewählt werden können, lassen sich in Boolescher Logik kodieren und mit Hilfe von effizienten Boolean Satisfiability Problem (SAT) Solvern lösen. Das Lösen von SAT Problemen ist ein schwieriges Problem (NP-vollständig) und die Kernfrage ist, ob variabilitätsgewahre

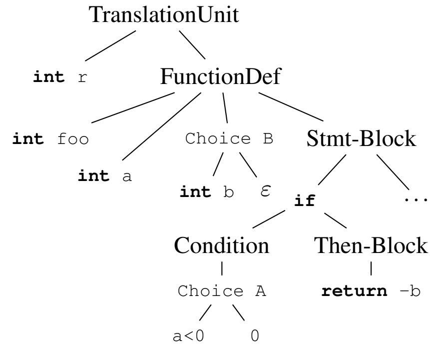
Analysen in der Praxis auch für Systeme mit tausenden von Konfigurationsoptionen skalieren. Für Laufzeitmessungen haben wir die Liveness-Berechnung als variabilitätsgewahre Analyse mit drei verbreiteten Sampling-Verfahren verglichen:

- single configuration: Das Verfahren beschränkt sich auf die Analyse einer einzelnen Systemvariante, in der möglichst viele Optionen ausgewählt sind.
- code coverage: Das Verfahren analysiert mehrere Systemvarianten, sodass jedes variable Codefragment mindestens einmal berücksichtigt wird [Ta12].
- pair-wise: Alle paarweisen Kombinationen von Konfigurationsoptionen werden analysiert, um Wechselwirkungen zwischen diesen zu bestimmen [Ku04].

Um die praktische Anwendung zu überprüfen, haben wir die Laufzeit beider Analyseansätze in den drei größeren Softwaresystemen BUSYBOX (eine Werkzeugsammlung für eingebettete Systeme), LINUX (ein open-source Betriebssystem) und OPENSSL (eine kryptografische Bibliothek) untersucht. Alle drei Systeme haben eine astronomisch große Anzahl an Systemvarianten. Beim Vergleich zeigte sich (Abbildung 2), dass der Aufwand für variabilitätsgewahre Analysen (gestrichelte Linie) nur wenig höher ist als die Analyse einiger weniger Systemvarianten. Der Break-even-Punkt, an dem variabilitätsgewahre Analysen schneller als Sampling-Verfahren sind, liegt laut unseren Messungen zwischen zwei bis drei Varianten.

```
 1 #ifdef A #define EXPR (a<0)
 2 #else    #define EXPR 0
 3 #endif
 4
 5 int r;
 6 int foo(int a #ifdef B, int b #endif) {
 7   if (EXPR) {
 8     return -b;
 9   }
10   int c = a;
11   if (c) {
12     c += a;
13     #ifdef B c += b; #endif
14   }
15   return c;
16 }
```

(a) Variable Codefragmente (Zeile 1 bis 3, 6, und 13); Annotationen sind zur einfacheren Darstellung in Codezeilen integriert.



(b) Ausschnitt des variablen AST; `Choice` kodiert eine statische Bedingung über Konfigurationsoptionen mit AST Knoten als Wert oder dem leeren Element ε.

Abbildung 1: Variabler Quellcode mit dem entsprechenden variablen AST.

Aufwand. Der größte Erfolgsfaktor variabilitätsgewahrer Analysen im Vergleich zu Sampling-Verfahren ist das Teilen von Analyseergebnissen zwischen verschiedenen Systemvarianten. Dieses Teilen stellt sicher, dass Gemeinsamkeiten und Unterschiede nur einmal berechnet werden müssen und somit redundante Berechnungen vermieden werden. Das Teilen von Ergebnissen während der Analyse wird durch drei Strukturmuster sichergestellt, die maßgeblich für die Effizienz variabilitätsgewahrer Analysen verantwortlich sind: 1) late splitting, 2) early joining und 3) local variability representation (Abbildung 3) [Li13]. Late splitting stellt sicher, dass eine Analyse sich erst beim Auftreten von Variabilität in der Eingabe in unterschiedliche Analysepfade aufspaltet und damit unterschiedliche Systemva-

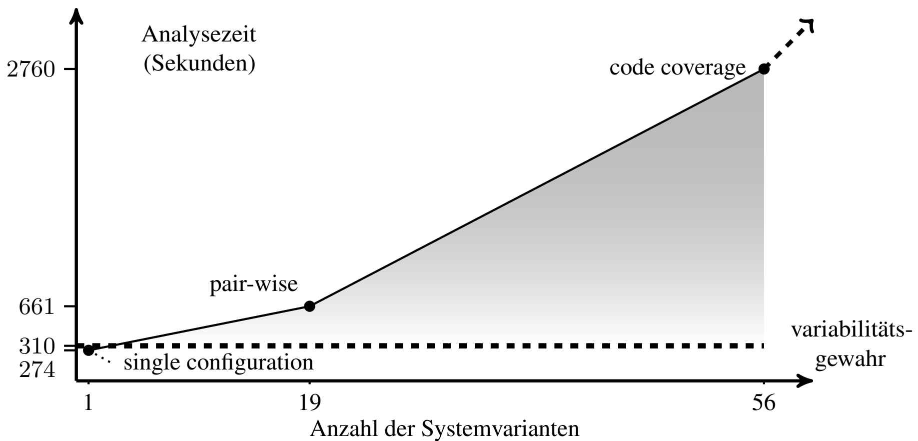


Abbildung 2: Anzahl der Varianten vs. Analysezeit (Liveness-Berechnung) in BUSYBOX.

rianten berücksichtigt. Early joining führt unterschiedliche Analysepfade zusammen wenn ein gemeinsamer Punkt in der Eingabe erreicht wird. Beide Muster dämmen das exponentielle Wachstum des Analyseaufwands ein, der durch lokale Variationen im Quellcode entsteht. Local variability representation stellt sicher, dass Analyseergebnisse redundanzfrei mit Variabilitätsinformationen gespeichert werden und all jenen Systemvarianten zugeordnet sind, denen sie entsprechend ihrer Konfiguration angehören.

Unsere Ergebnisse zeigen, dass die Berechnung der repräsentativen Mengen in Sampling-Verfahren nicht unerheblich ist. Je nach verwendetem Verfahren sind die zugrundeliegenden Algorithmen in der Komplexitätsklasse NP-vollständig, sodass Sampling-Berechnungen für Systeme mit einer großen Anzahl an Konfigurationsoptionen nicht mehr skalieren. Für zwei Sampling-Verfahren übersteigt der Aufwand zur Berechnung der repräsentativen Mengen den Aufwand für die eigentliche Analyse. Darüber hinaus zeigen unsere Ergebnisse, dass die Analysezeit für variabilitätsgewahren Analysen teilweise sogar unter den Berechnung der repräsentativen Mengen liegt. Während dafür mitunter sehr leistungsfähige Computer mit hoher Speicherausstattung notwendig sind, reicht für die Anwendung variabilitätsgewahrer Analysen ein Standardcomputer mit 2 bis 8 GB Arbeitsspeicher zur Berechnung aus.

Fehlererfassung. Auf der Grundlage der in Abbildung 3 genannten Strukturmuster haben wir ein intra-prozedurales Datenfluss-Framework entwickelt, das die Entwicklung variabilitätsgewahrer Analysen für typische Programmierfehler vereinfacht. Zu den Fehlern gehören beispielsweise die fehlende Initialisierung von Variablen und die doppelte Freigabe von dynamisch angefordertem Speicher. Beide Fehler sind sehr ernst zu nehmen, da sie in der Praxis zu einem undefinierten Systemverhalten führen können oder das Potenzial für Angriffe auf die Systemsicherheit bergen. Im Vergleich zwischen variabilitätsgewahrer Analyse und Sampling dient ersterer Analyseansatz als Referenz, da alle gültigen Varianten hinsichtlich möglicher Programmierfehler analysiert werden. Unserer Ergebnisse zeigen, dass Sampling-Verfahren nicht alle Fehler in allen gültigen Varianten eines konfigurierbaren Systems finden. Sampling-Verfahren finden zwischen 14 % und 99 % aller Fehler, die durch variabilitätsgewahre Analysen aufgespürt werden. Dies legt nahe, dass für eine umfassende

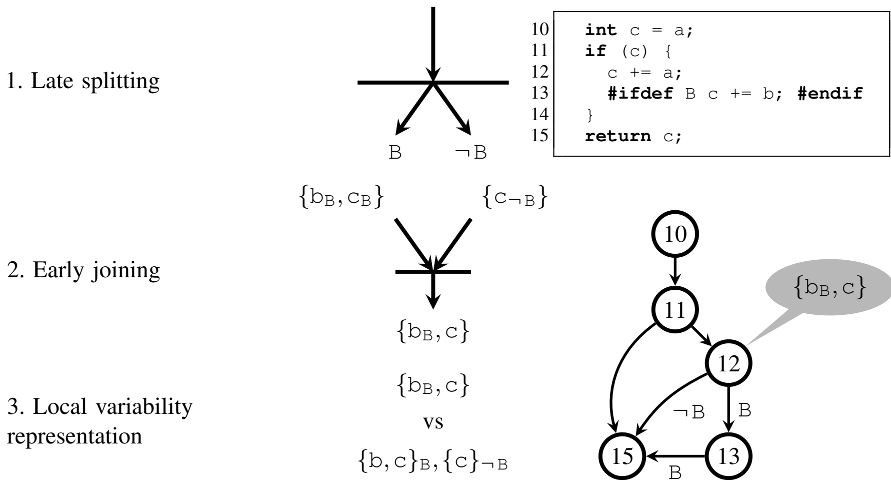


Abbildung 3: Strukturmuster variabilitätsgewahrer Analysen am Beispiel von der Liveness-Berechnung für einen Ausschnitt der Abbildung 1; rechts im Bild ist ein variabler CFG mit dem Ergebnis der Liveness-Berechnung (callout) für die Anweisung in Zeile 12; $\mathtt{b}_{\mathtt{B}}$ bedeutet, dass die Variable `b` nur dann Bestandteil des Analyseergebnisses ist, wenn die Konfigurationsoption `B` ausgewählt wurde (Lesezugriff in Zeile 13); Variable `c` ist in den Analyseergebnissen aller Systemvarianten enthalten (Lesezugriffe in den Zeilen 13 und 15).

Analyse aller Varianten variabilitätsgewahre Analysen verwendet werden sollten. Denn Tatsache ist, dass Programmierfehler gerade in jenen Varianten auftreten könnten, die nicht durch das verwendete Sampling-Verfahren analysiert wurden. In der Folge ist das Aufspüren von Fehlern in unterschiedlichen Varianten eingeschränkt und je nach verwendetem Sampling-Verfahren nicht systematisch. Obwohl unsere variabilitätsgewahren Analysen nicht die Reife existierender Werkzeuge erreichen und möglicherweise einige „false positives“ enthalten,[1] so konnten wir bereits gefundene und von Konfigurationsoptionen abhängige Programmierfehler in LINUX [Ab14] nachstellen und aufspüren.

4 Refactoring C Code unter Berücksichtigung von #ifdefs

Die größte Herausforderung beim Refactoring konfigurierbarer Systeme ist die Sicherstellung das sich das Systemverhalten nach Anwendung des Refactorings nicht ändert. Insbesondere die Disziplin von **`#ifdef`** Annotationen und die hohe Variantenvielfalt existierender Systeme sind in der Praxis dabei die größten Probleme [Li15b]. Bei einer Untersuchung 17 verschiedener Transformationswerkzeuge (darunter Entwicklungsumgebungen wie beispielsweise ECLIPSE) zeigten sich die folgenden Probleme:

- Refactorings sind nur mittels einfacher Editor-Funktionen („Suche und Ersetze“) möglich, die keinen adäquaten Ersatz für ein Transformationswerkzeug bieten.

[1] Das Auftreten von „false positives“ ist ein generelles Problem statischer Analysen und unabhängig von variabilitätsgewahren Techniken. Ihr Vorkommen lässt sich allein durch verbesserte Fehlerberechnungen, bspw. durch Pointer-Analysen, und die Ausweitung auf inter-prozeduralen Analysen eingrenzen.

- Die Werkzeuge unterstützen nur eine einzelne Systemvariante, sodass andere Systemvarianten nach der Anwendung von Refactorings Fehler enthalten könnten.
- Abhängigkeiten zwischen Konfigurationsoptionen werden nicht berücksichtigt und daher können einzelne Systemvarianten nach der Transformation fehlerbehaftet sein.
- Vereinzelt generieren Werkzeuge intern alle vom Refactoring betroffenen Systemvarianten und wenden Transformationen individuell an. Dieser Ansatz skaliert in der Praxis nicht für Systeme mit einer großen Anzahl an Systemvarianten.
- Durch den Einsatz von Heuristiken werden Garantien hinsichtlich der Sicherstellung des Systemverhaltens eingebüßt.

Abbildung 4 zeigt ein Beispiel für eine fehlerhafte Ausführung des EXTRACT-FUNCTION Refactorings in ECLIPSE. Die markierten Anweisungen in der Abbildung 4a sollen durch das Refactoring in eine eigene Funktion namens `foo` ausgelagert werden. Nach der Quellcodetransformation stellt sich heraus, dass sich das Verhalten durch die Änderung der Anweisungsfolge geändert hat. Solche Fehler sind schwer zu finden, gerade weil die Ausführung des Refactorings ohne weitere Rückmeldung in ECLIPSE erfolgte.

```
1  #include <stdio.h>
2  #define DEBUG 1
3
4  int main() {
5    if (DEBUG) {
6      printf("Debug mode entered.\n");
7  #ifdef A
8      printf("Option A enabled.\n");
9  #endif
10     printf("Debug mode left.\n");
11   }
12   return 0;
13 }
```

Ausgabe der Variante A:

```
1 Debug mode entered.
2 Option A enabled.
3 Debug mode left.
```

(a) Vor EXTRACT FUNCTION `foo`

```
1  #include <stdio.h>
2  #define DEBUG 1
3
4  void foo() {
5    printf("Debug mode entered.\n");
6    printf("Debug mode left.\n");
7  }
8
9  int main() {
10   if (DEBUG) {
11     foo();
12 #ifdef A
13     printf("Option A enabled.\n");
14 #endif
15   }
16   return 0;
17 }
```

Ausgabe der Variante A:

```
1 Debug mode entered.
2 Debug mode left.
3 Option A enabled.
```

(b) Nach EXTRACT FUNCTION `foo`

Abbildung 4: Ein Beispiel für die fehlerhafte Anwendung eines EXTRACT-FUNCTION Refactorings in ECLIPSE. Die Programmausgaben vor und nach dem Refactoring zeigen die Änderung des Systemverhaltens.

Um die Einschränkungen bestehender Ansätze und Transformationswerkzeuge zu bewältigen, haben wir ein eigenes Transformationswerkzeug namens MORPHEUS entwickelt. MORPHEUS nutzt variabilitätsgewahre Abstraktionen von ASTs und CFGs zur Sicherstellung des Systemverhaltens bei Refactorings. Dazu nutzen wir die gleiche Infrastruktur wie zuvor bei der Entwicklung von Analysewerkzeugen (Abschnitt 3). Die Nutzung variabilitätsgewahrer Abstraktionen und Algorithmen stellt sicher, dass Transformationen mit

Hinblick auf alle Systemvarianten sowohl vollständig als auch performant sind. MORPHEUS implementiert drei weit verbreitete Standard Refactorings:

- RENAME IDENTIFIER zielt auf die konsistente Umbenennung von Symbolnamen ab (z.B. Funktionen, Variablennamen und selbst definierte Datentypen).
- EXTRACT FUNCTION: ein Codefragment zusammengehöriger Anweisungen wird in eine eigene Funktion ausgelagert und durch einen Funktionsaufruf ersetzt.
- INLINE FUNCTION ist die Umkehrfunktion zu EXTRACT FUNCTION. Sie ersetzt einen Funktionsaufruf durch die entsprechende Funktionsimplementierung.

Um zu demonstrieren, dass variabilitätsgewahre Transformationen in der Praxis funktionieren, haben wir MORPHEUS für Refactorings in drei realen Systemen eingesetzt. Neben den beiden Systemen BUSYBOX und OPENSSL aus Abschnitt 3 setzen wir auch das eingebettete Datenmanagementsystem SQLITE ein. In 11 479 unterschiedlichen Refactorings konnten wir zeigen, dass MORPHEUS effizient arbeitet (einzelne Refactorings brauchen weniger als eine Sekunde) und auch für größere Softwaresysteme skaliert. Wie schon zuvor bei der Entwicklung variabilitätsgewahrer Analysewerkzeuge stellt auch hier der Einsatz von effizienten SAT Solvern kein Problem dar. Um zu überprüfen, dass variabilitätsgewahre Transformationen keine Verhaltensänderungen verursachen, haben wir eine Standardmethode für das Testen von Transformationswerkzeugen erweitert. Dazu wurden für jedes Refactoring alle betroffenen Systemvarianten ermittelt und ihre Äquivalenz durch einen vorher-nachher Vergleich von Testergebnissen bestimmt.

5 Zusammenfassung

Diese Arbeit schließt eine Lücke in der Entwicklung konfigurierbarer Systeme durch die Bereitstellung skalierbarer Techniken und Werkzeuge für die Analyse und Transformation großer, konfigurierbarer Softwaresysteme. Die entwickelten Techniken können als Ausgangspunkt für weitere Forschungsarbeiten im Gebiet der Analyse und Transformation konfigurierbarer Systeme dienen. Die in der Arbeit vorgestellten Werkzeuge eignen sich besonders dafür, da sie bereits mehrfach mit realen Systemen (z.B. LINUX Kernel) eingesetzt wurden. Abgesehen von der Forschung zeigen die erzielten Ergebnisse Wege auf, wie existierende Werkzeuge (z.B. ECLIPSE) verbessert werden können. Davon würden sowohl Softwareentwickler als auch Anwender profitieren, da sich konfigurierbare Systeme schneller entwickeln lassen und weniger Fehlern aufweisen. Schließlich lassen sich konfigurierbare Systeme mit verbesserten Analyse- und Transformationswerkzeugen nun ähnlich leicht entwickeln, wie es für Einzelsysteme schon seit langem gängige Praxis ist. Dies gilt für konfigurierbare Systeme im Allgemeinen und für Präprozessor-basierte Systeme im Speziellen.

Literatur

[Ab14] Abal, I. et al.: 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In: Proc. Int'l Conf. Automated Softw. Eng. (ASE). ACM, S. 421–432, 2014.

[Fa96] Favre, J.-M.: Preprocessors from an Abstract Point of View. In: Proc. Int'l Conf. Softw. Maintenance (ICSM). IEEE, S. 329–339, 1996.

[Kä09] Kästner, C. et al.: Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In: Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE). Springer, S. 174–194, 2009.

[Kä11] Kästner, C. et al.: Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In: Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM, S. 805–824, 2011.

[KM03] Ko, A.; Myers, B.: Development and Evaluation of a Model of Programming Errors. In: Proc. Symp. Human Centric Computing Languages and Environments (HCC). IEEE, S. 7–14, 2003.

[Ku04] Kuhn, D. et al.: Software Fault Interactions and Implications for Software Testing. IEEE Trans. Softw. Eng., 30(6):418–421, 2004.

[Li10] Liebig, J. et al.: An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In: Proc. Int'l Conf. Softw. Eng. (ICSE). ACM, S. 105–114, 2010.

[Li11] Liebig, J. et al.: Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In: Proc. Int'l Conf. Aspect-Oriented Softw. Dev. (AOSD). ACM, S. 191–202, 2011.

[Li13] Liebig, J. et al.: Scalable Analysis of Variable Software. In: Proc. European Softw. Eng. Conf. and Symp. Found. Softw. Eng. (ESEC/FSE). ACM, S. 81–91, 2013.

[Li15a] Liebig, J.: Analysis and Transformation of Configurable Systems. Dissertation, University of Passau, 2015.

[Li15b] Liebig, J. et al.: Morpheus: Variability-Aware Refactoring in the Wild. In: Proc. Int'l Conf. Softw. Eng. (ICSE). ACM, S. 380–391, 2015.

[SC92] Spencer, H.; Collyer: #ifdef Considered Harmful, or Portability Experience with C News. In: Proc. USENIX Tech. Conf. USENIX Assoc., S. 185–197, 1992.

[Ta12] Tartler, R. et al.: Configuration Coverage in the Analysis of Large-scale System Software. SIGOPS Operating Systems Review, 45(3):10–14, 2012.

[Th14] Thüm, T. et al.: A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Computing Surveys, 47(1):6:1–6:45, 2014.

[Wa14] Walkingshaw, E. et al.: Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In: Proc. Int'l Symp. New Ideas in Programming and Reflections on Softw. (Onward!). ACM, S. 213–226, 2014.

Jörg Liebig wurde 1983 in Wolfen geboren. Er begann sein Studium in Ingenieurinformatik an der Otto-von-Guericke-Universität Magdeburg im Herbst 2003 und beschäftigte sich bereits in seiner Diplomarbeit mit der Programmierung konfigurierbarer Systeme. Nach seinem Abschluss im Dezember 2008 begann er sein Promotionsstudium als wissenschaftlicher Mitarbeiter am Lehrstuhl für Programmierung (Prof. Christian Lengauer, Ph.D.) und später am Lehrstuhl für Softwareproduktlinien (Prof. Dr. Sven Apel). Der Doktorgrad wurde ihm im April 2015 in Passau „summa cum laude“ verliehen. Im Anschluss wechselte Dr. Liebig zur Firma Method Park und berät nun Unternehmen bei der Entwicklung und Handhabung konfigurierbarer Systeme.