Analysestrategien für konfigurierbare Systeme[1]

Alexander von Rhein[2]

Abstract: Viele moderne Softwaresysteme, wie z.B. Linux, lassen sich mit Tausenden von Konfigurationsoptionen anpassen. Mit Hilfe dieser Optionen kann eine oft astronomisch große Anzahl unterschiedlicher Systemvarianten hergestellt werden. Diese Variantenvielfalt bringt enorme Herausforderungen hinsichtlich der Korrektheit der Systemvarianten mit sich. In dieser Arbeit entwickeln und untersuchen wir Techniken zur Analyse von variantenreichen Systemen. Insbesondere vergleichen wir drei Ansätze zum Design solcher Analysen konzeptionell und praktisch (z.B. hinsichtlich ihrer Laufzeit und der Vollständigkeit ihrer Ergebnisse).

1 Einleitung

Ein konfigurierbares System ermöglicht es, individuelle Systemvarianten anhand einer Auswahl von Konfigurationsoptionen abzuleiten. Konfigurationsoptionen werden in der Praxis, zum Beispiel, mit Präprozessorannotationen (#ifdefs) implementiert. Um bei der Systemanalyse mit der oft sehr großen Anzahl möglicher Konfigurationen fertig zu werden, implementieren verschiedene Analyseansätze (z.B. zur Verifikation von konfigurierbaren Systemen) verschiedene Strategien, um die Konfigurierbarkeit zu berücksichtigen.

Eine einfache Strategie, die in der Praxis häufig angewendet wird, besteht darin, nur eine Teilmenge aller Systemvarianten zu verwenden. Während diese *Stichproben Strategie* den Analyseaufwand deutlich reduziert, sind die erhaltenen Informationen notwendigerweise unvollständig, da einige Varianten nicht analysiert werden. Eine zweite Strategie besteht darin, die gemeinsamen und die optionalen Komponenten eines konfigurierbaren Systems zu identifizieren und jede Komponente separat zu analysieren (diese Strategie wird auch als *featurebasierte* Strategie bezeichnet). Als dritte Strategie wurde in den letzten Jahren die *familienbasierte* Strategie entwickelt. Familienbasierte Ansätze analysieren die Codebasis eines konfigurierbaren Systems als Ganzes und nicht einzelne Varianten oder Teile des Systems. So können Gemeinsamkeiten einzelner Varianten genutzt werden um den Analyseaufwand zu reduzieren. Jede dieser drei Strategien hat Vor- und Nachteile, die sogar ihre Anwendung verhindern können (z.B. benötigt die familienbasierte Strategie typischerweise viel Arbeitsspeicher).

Das Ziel dieser Dissertation ist es, die effiziente Analyse von konfigurierbaren Systemen zu ermöglichen, selbst wenn die oben genannten Strategien nicht anwendbar

[1] Englischer Titel der Dissertation: “Analysis Strategies for Configurable Systems” [Rh16]
[2] CQSE GmbH, rhein@cqse.eu

sind (falls z.B. die familienbasierte Strategie aufgrund von Speicherbeschränkungen nicht anwendbar ist). Zu diesem Zweck haben wir ein Modell entwickelt, das die Schlüsselaspekte der Analysestrategien für konfigurierbare Systeme unabhängig von ihrer Implementierung und den Analysetechniken (z. B. Typprüfung, Modelchecking und Taint-Propagation) modelliert. Anhand dieses Modells haben wir eine Reihe von Analysestrategien für konfigurierbare Systeme entwickelt und implementiert. Um mehr über Vor- und Nachteile einzelner Strategien zu lernen, haben wir diese Analysestrategien in einer Reihe empirischer Studien verglichen.

Eines unserer wichtigsten Ergebnisse ist, dass familienbasierte Analysen weniger Ausführungszeit als die meisten stichprobenbasierten Analysen benötigen und gleichzeitig in der Lage sind, definitive Aussagen über alle Varianten eines konfigurierbaren Systems zu machen. Darüber hinaus identifizierten wir die Vor- und Nachteile von Analysestrategien und deren Vermeidung durch die Kombination von Strategien.

Wir haben zwei Schlüsselprobleme, die in Analysen für konfigurierbare Systeme auftreten, identifiziert und haben Unterstützungstechniken entwickelt, um diese zu lösen. Diese Techniken sind allgemein und gelten über unsere Forschung hinaus. Die entwickelten Techniken sind eine Methode zur Vereinfachung von Präsenzbedingungen (*presence-condition simplification*) und Methoden zur Kodierung von Variabilität (*variability encoding*). Die Vereinfachung der Präsenzbedingungen bietet eine einfache Methode, um die Größe der Ausgabe oder die interne Datenstruktur von Analysen zu reduzieren. In dieser Zusammenfassung gehen wir nicht näher auf presence-condition simplification ein. Wir verweisen auf die veröffentlichte Dissertation [Rh16]. Variabilitätskodierung ist eine Programmtransformation zur Umwandlung von Variabilität zur Übersetzungszeit in Variabilität zur Laufzeit bereit, was viele familienbasierte Analysen ermöglicht (siehe Abschnitt 2).

Konfigurierbare Systeme sind Systeme, die maßgeschneidert werden können, um verschiedene Rollen in unterschiedlichen Umgebungen und Anwendungsszenarien zu erfüllen. Konfigurationsoptionen oder “Features” definieren Unterschiede zwischen Systemvarianten.

Zur Illustration von konfigurierbaren Systemen stellen wir ein konfigurierbares Druckersystem vor. In der Praxis wurde ein ähnliches, größeres konfigurierbares System für Hewlett-Packard Drucker von der Owen Firmware Cooperative implementiert. Das Druckersystem besteht aus Features, die unterschiedliche Druck- und Scanfunktionen implementieren. Varianten des Systems können Treiber für eine Familie von Druckern darstellen. Einige der Varianten bieten Funktionalitäten wie Scannen oder Duplexdruck, und andere nicht (z.B. zur Preisreduzierung).

Das konfigurierbare Drucksystem verfügt über vier Features: *BasicPrinter*, *Duplex*, *Scan* und *Copy*. *BasicPrinter* bietet zentrale Druckfunktionen wie z.B. einseitiges Drucken, eine Benutzeroberfläche und Protokolle für den Druckeranschluss. Das Feature *Duplex* ermöglicht das automatische Drucken von Duplexseiten. *Scan*

```
1  class Printer {
2    void printMulti
3      (Page front, Page back) {
4      ... // print both pages on one sheet
5    }
6    #ifdef Duplex
7    void printDuplex
8      (Page front, Page back) {
9      ... // duplex printing
10   }
11   #endif
12   void print(Page front, Page back) {
13   #ifdef Duplex
14     printDuplex(front, back);
15   #else
16     printMulti(front, back);
17   #endif
18   }
19 }
```

(a) #ifdef Implementierung

```
1  class Printer {
2    void printMulti
3      (Page front, Page back) {
4      ... // print both pages on one sheet
5    }
6
7    void printDuplex
8      (Page front, Page back) {
9      ... // duplex printing
10   }
11
12   void print(Page front, Page back) {
13
14     printDuplex(front, back);
15
16
17
18   }
19 }
```

(b) Systemvariante mit den Optionen *BasicPrinter* und *Duplex*

Abb. 1: Eine ifdef-basierte Implementierung und eine Variante des Druckersystems. Leerzeilen in der Variante zeigen wo Code durch #ifdefs weggefallen ist.

ermöglicht es Benutzern, die Scan-Hardware einiger Drucker zu verwenden. *Copy* (erfordert das Feature *Scan*) ermöglicht es Benutzern, Dokumente zu duplizieren.

Abbildung 1a zeigt einen Ausschnitt einer Implementierung mit #ifdef Annotationen. Zur Kompillierzeit können die in den #ifdefs benutzten Optionen mit Werten belegt werden und so Varianten generiert werden. Abbildung 1b zeigt eine Variante.

Für unsere Experimente haben wir sowohl sehr große Systeme, wie den Linux Kernel oder das Datenbanksystem SQLite, als auch kleinere Systeme genutzt. Aufwändige Analysen, wie zum Beispiel Modelchecking, können nicht ohne weiteres auf großen Softwaresystemen angewandt werden. Um unsere Analysen in realistischem Kontext zu evaluieren, haben wir Entwürfe für konfigurierbare Systeme, die von anderen Forschern veröffentlicht wurden, genutzt und entsprechende Programme in C und Java implementiert. Diese Systementwürfe sind in der Forschungsgemeinschaft akzeptiert und werden oft verwendet (z.B. [Cl11; SRA13]). Unsere Implementierungen wurden mittlerweile als Benchmarks von anderen Forschern verwendet (z.B. [Be15]), so dass sie selbst einen Beitrag zur Forschung darstellen.

2 Variabilitätskodierung

Bevor wir verschiedene Strategien für die Analyse von konfigurierbaren Systemen bewerten, müssen wir Mittel entwickeln, um die Strategien umzusetzen. Für die variantenbasierte Strategie ist nahezu kein Aufwand erforderlich, da die Analyseobjekte (Varianten) normale, nicht konfigurierbare Systeme sind. Solche Systeme können oft mit handelsüblichen Werkzeugen analysiert werden. Für die stichprobenbasierte

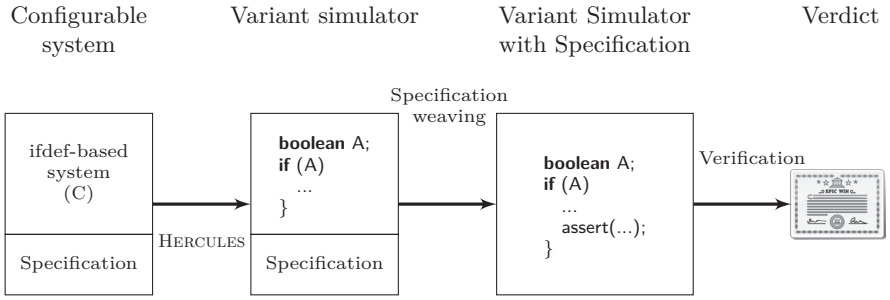

Abb. 2: Variabilitäts-gewahre Verifikation.

Strategie müssen geeignete Stichproben generiert werden. Dieses Thema wurde bereits von anderen Forschern ausführlich erörtert (z.B. [NL11; OMR10]). Die verbleibende Aufgabe, die wir hier ansprechen, ist die Umsetzung der familienbasierten Strategie.

Dafür stellen wir *Variabilitätskodierung* vor, eine Programmtransformation, die aus einem konfigurierbaren System einen *Variantensimulator* erzeugt. Die Transformation kodiert die Variabilität des konfigurierbaren Systems, die normalerweise zur Kompillierzeit aufgelöst wird, als dynamische Variabilität in einem Variantensimulator. Eine wichtige Eigenschaft der Variabilitätskodierung ist, dass der resultierende Simulator das Verhalten einer beliebigen Variante (auf einer abstrakten Ebene) simulieren kann. Durch diese *Verhaltenserhaltung* können wir Simulatoren in familienbasierten Analysen von konfigurierbaren Systemen einsetzen und auf Basis der Analyseergebnisse Aussagen über alle Varianten treffen (Abschnitt 3).

Bei der Transformation werden Konfigurationsoptionen mit globalen Programmvariablen kodiert, und statische Ausführungsbedingungen (z.B. #ifdefs) werden mit bedingten Anweisungen in der Zielsprache (if-Anweisungen) kodiert. Classen et al. [Cl11] und Post und Sinz [PS08] haben manuelle Ansätze verwendet, die der Variabilitätskodierung ähnlich sind. Unser Ansatz zur Variabilitätskodierung ist *automatisch* und kann auf große Codebasen angewendet werden.

Anwendungsszenario Hier beschreiben wir eine Anwendung der Variabilitätskodierung unter Verwendung von Beispielcode aus dem Linux Kernel. Insbesondere verwenden wir Code aus der Variability Bug Database, einer Sammlung von realen Bugs, die im Linux Kernel gefunden wurden und nur in bestimmten Konfigurationen vorkommen. In diesem Szenario nutzen wir Modelchecking um (vereinfachte) Programme aus der Bug-Datenbank zu verifizieren. Wir können zeigen, dass Modelchecking genau die Konfigurationen als fehlerhaft identifiziert, die auch in der Datenbank dokumentiert sind. Der Fehler Bug1[3] (Abb. 3a) verursacht einen Aufruf des Linux `BUG()` Makros, wenn die Konfigurationsoption `VLAN_8021Q` deaktiviert

[3] `http://vbdb.itu.dk/#bug/linux/d549f55`

```
1  #include <assert.h>
2  #include <stdbool.h>
3  #include <stdlib.h>
4  #ifdef CONFIG_VLAN_8021Q
5  void* vlan_dev_real_dev() {
6    return NULL;
7  }
8  #else
9  void* vlan_dev_real_dev() {
10     assert(false); // (3) ERROR
11     return NULL;
12 }
13 #endif
14 #if defined(CONFIG_IPV6) ||
15     defined(CONFIG_VLAN_8021Q)
16 static int ocrdma_inet6addr_event() {
17   vlan_dev_real_dev(); // (2)
18   return 0;
19 }
20 #endif /* IPV6 and VLAN */
21 int main(int argc, char** argv) {
22 #if defined(CONFIG_IPV6) ||
23     defined(CONFIG_VLAN_8021Q)
24   ocrdma_inet6addr_event(); // (1)
25 #endif
26   return 0;
27 }
```

(a) Ursprünglicher Code von Bug1

```
1  ...
2  int id2i_config_ipv6;
3  int id2i_config_vlan_8021q;
4  extern int __VERIFIER_nondet_int();
5  void id2i_init() {
6    id2i_config_vlan_8021q =
7      __VERIFIER_nondet_int();
8    id2i_config_ipv6 =
9      __VERIFIER_nondet_int();
10 }
11 void *_5_vlan_dev_real_dev() {
12   return ((void *) 0);
13 }
14 void *_6_vlan_dev_real_dev() {
15   (0 ? ((void ) 0) : __assert_fail(...);
16   return ((void *) 0);
17 }
18 static int ocrdma_inet6addr_event() {
19   if (((id2i_config_vlan_8021q ) )) {
20     _5_vlan_dev_real_dev();
21   }
22   if ((((! id2i_config_vlan_8021q) ) )) {
23     _6_vlan_dev_real_dev();
24   }
25   return 0;
26 }
27 int main(int argc , char **argv ) {
28   id2i_init();
29   if ((((! id2i_config_vlan_8021q) &&
30         id2i_config_ipv6) ||
31        id2i_config_vlan_8021q)) {
32     ocrdma_inet6addr_event();
33   }
34   return 0;
35 }
```

(b) Code des Variantensimulators für Bug1

Abb. 3: Ursprünglicher und variabilitätskodierter Quellcode (erzeugt mit unserem Tool Hercules) von Bug1. Der Programmpfad, der den Fehler verursacht, wird mit Kommentaren im ursprünglichen Code kommentiert. Vertikale Balken zeigen, welche Features und Anweisungen im Originalcode (links) zu welchen Features und Anweisungen im Simulator (rechts) gehören.

ist und die Option IPV6 aktiviert ist. Daher ist die Präsenzbedingung für den Fehler IPV6 $\wedge \neg$VLAN _8021Q.

Abbildung 2 zeigt den Ablauf der Verifikation eines konfigurierbaren Systems. Wir verwenden unser Tool Hercules, um die #ifdef-Variabilität in Laufzeit-Variabilität umzuwandeln (Abb. 3b). Im nächsten Schritt fügen wir die Systemspezifikation in den Code ein. In diesem Szenario ist die Spezifikation (assert(false) wird nicht erreicht) bereits im Code vorhanden. Dann verifizieren wir den Code mit einem Model Checker (siehe Abschnitt 3).

Bei der Variabilitätskodierung werden Konfigurationsoptionen mit globalen Variablen (*Featurevariablen*) dargestellt. Die Featurevariablen werden mit Rückgabewerten der speziellen Funktion __VERIFIER_nondet_int() initialisiert. Es wird da-

von ausgegangen, dass diese Funktion einen beliebigen Wert vom Typ int zurückgibt [Be15]. In unserem Fall bedeutet diese Annahme, dass alle vier Kombinationen von id2i_config_ipv6 == 0 oder id2i_config_ipv6! = 0 und id2i_config_vlan_8021q == 0 oder id2i_config_vlan_8021q! = 0 im Programm erreichbar sind. Dies entspricht der Abdeckung aller gültigen Konfigurationen des ursprünglichen konfigurierbaren Systems.

Für die Überprüfung des generierten Programms haben wir den Modelchecker CPAchecker erweitert. Die Erweiterungen werden im Abschnitt 3 näher beschrieben. Sie ermöglichen CPAchecker, die Programmausführung in allen gültigen Konfigurationen zu prüfen und ein zusammengefasstes Ergebnis auszugeben. Nach der Überprüfung von Bug1 gibt CPAchecker folgende Fehlerbedingung aus: !Id2i_config_vlan_8021q@0 id2i_config_ipv6@0. Diese entspricht genau der in der Bug Datenbank angegebenen Bedingung $\mathsf{IPV6} \wedge \neg \mathsf{VLAN\_8021Q}$.

Diese Illustration zeigt, dass die Kombination von Variabilitätskodierung und Modelchecking effektiv verwendet werden kann, um Fehler in konfigurierbaren Programmen zu identifizieren und präzise Fehlerbedingungen zu berichten.

Die Notwendigkeit eines formalen Korrektheitsbeweises Variabilitätskodierung wurde in mehreren Forschungsprojekten eingesetzt und ermöglichte beträchtliche Analysebeschleunigungen im Vergleich zur variantenbasierten Analyse [Ap11; Ap13; SRA13]. In diesen Projekten und im vorherigen Beispiel verwendeten die Subjektsysteme einfache Sprachkonstrukte, so dass wir sicher davon ausgehen konnten, dass der generierte Code das Verhalten des ursprünglichen Programms erhalten hatte und dass die nachfolgenden Analysen gültig waren. Es war jedoch unklar, ob Variabilitätskodierung in Gegenwart von anderen Sprachfunktionen, wie Vererbung oder switch Anweisungen umzusetzen ist. Variabilitätskodierung ist eine komplexe Programmtransformation, und in der Praxis können sogar relativ einfache automatische Refaktorisierungs-Werkzeuge das Verhalten bei Vorhandensein von statischen Konfigurationsoptionen nicht erhalten.

Um unser Verständnis der Variabilitätskodierung zu verbessern, haben wir die Verhaltenserhaltung in der Variabilitätskodierung formal untersucht. Wir haben die Variabilitätskodierung, basierend auf einer kleinen, formalen Sprache definiert und formal bewiesen, dass Verhaltenserhaltung gilt. Für den Beweis haben wir den Kontrollfluss von (beliebigen) Systemvarianten und entsprechenden Variantensimulatoren als Transitionssysteme modelliert. Auf diesen Transitionssystemen haben wir schwache Bisimulation bewiesen und so gezeigt, dass das Verhalten der Varianten korrekt von Variantensimulatoren simuliert wird. Außerdem haben wir diskutiert, wie weitergehende Sprachmerkmale behandelt werden können (z.B. Vererbung).

3 Familienbasiertes Modelchecking

In diesem Abschnitt stellen wir unseren Ansatz zum effizienten, familienbasierten Modelchecking von konfigurierbaren Systemen vor. Unser neuer Ansatz basiert

auf einem modularen, zweistufigen Verfahren: Zuerst verwenden wir Variabilitätskodierung, um Varianten-Simulatoren zu erzeugen. Im zweiten Schritt wird ein Modelchecking-Werkzeug eingesetzt, um zu prüfen, ob die generierten Simulatoren alle Spezifikationen der jeweiligen Systeme erfüllen. Wenn das Werkzeug diese Eigenschaft beweisen kann, erfüllen alle Varianten auch ihre Spezifikationen (da die Variabilitätskodierung das Variantenverhalten erhält).

Für Modelchecking von Simulatoren könnten wir existierende Modelchecking Werkzeuge einsetzen, die, mit genügend Zeit und Speicher, gültige Resultate liefern würden. Allerdings können wir die Effizienz mit *variabilitätsgewahren* Optimierungen noch verbessern. Diese Optimierungen nutzen das Wissen über die Kodierung von Variabilität in Simulatoren.

Algorithmus 1: Algorithmus mit Pfadvereinigung und Variabilitätseinschränkung.

```
   Data: initial state q_0
   Result: Set reached_states
1  global-constraint := true
2  add(reached_states, q_0)
3  enqueue(wait_queue, q_0)
4  while ¬ empty(wait_queue) do
5      let v := dequeue(wait_queue)
6      if hasNoPropertyViolations(v) then
7          foreach w ∈ direct_succ(v) do
8              if sat(w_pc ∧ global-constraint) then
9                  if ¬∃ z ∈ reached_states : z_e = w_e then
10                     add(reached_states, w)
11                     enqueue(wait_queue, w)
12                 else
13                     let z ∈ reached_states with z_e = w_e
14                     if z_pc ⇏ w_pc then
15                         // new state is not completely
                           covered by existing state
16                         let z_pc := w_pc ∨ z_pc
17                         if z ∉ wait_queue then
18                             // explore the state again
19                             enqueue(wait_queue, z)
20                         end
21                     end
22                 end
23             end
24         end
25     else
26         global-constraint := global-constraint ∧ ¬v_pc
27     end
28 end
29 return reached_states
```

Beim Überprüfen eines Programms konstruiert ein Modelchecker einen Erreichbarkeitsgraphen, in dem jeder Knoten einen konkreten Zustand darstellt, den das Programm während der Ausführung erreichen kann. Falls einer der erreichbaren Zustände ein Fehlerzustand ist, wird ein Fehler im Programm erkannt. In *explizitem Modelchecking* werden konkrete Variablenwerte in den Zuständen gespeichert. Algorithmus 1 (ohne die farbigen Rechtecke) zeigt einen einfachen Breitensuche-Algorithmus für den Aufbau und die Prüfung eines solchen Erreichbarkeitsgraphen.

Unsere Optimierung besteht aus drei Schritten: (1) Einführung von Präsenzbedingungen, (2) Pfadvereinigung und (3) Variabilitätseinschränkung. Die Optimierungen führen zu einer Verkleinerung des generierten Zustandsraumes und damit zur Beschleunigung der Analyse (*wobei die Korrektheit und Vollständigkeit der Ergebnisse erhalten bleibt*).

Als ersten Schritt führen wir Präsenzbedingungen für Zustände im Erreichbarkeitsgraphen ein. Eine Präsenzbedingung ist eine aussagenlogische Formel über Konfigurationsoptionen, die bestimmt, in welchen Systemvarianten ein Zustand erreichbar ist. Die Nutzung von Präsenzbedingungen ermöglicht uns den nächsten Optimierungsschritt.

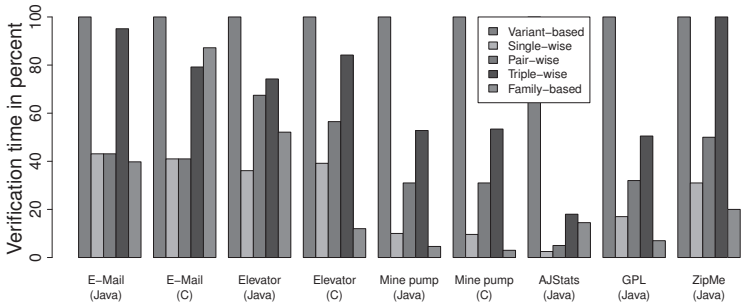

Abb. 4: Vergleich der Laufzeiten für die Verifikation der Systeme; Die Laufzeit der variantenbasierten Strategie definiert den 100 % Wert

Die Pfadvereinigung erlaubt es uns gleiche Zustände, die in verschiedenen Varianten vorkommen, gemeinsam zu behandeln. Das violette Rechteck in Algorithmus 1 zeigt die erforderliche Änderung. Wir erlauben dem Algorithmus verschiedene erreichbare Zustände als einen Zustand zu behandeln, wenn sie den gleichen Programmzustand repräsentieren (selbst wenn sie in unterschiedlichen Varianten vorkommen).

Die dritte Optimierung, Variabilitätseinschränkung, verhindert, dass Teile des Erreichbarkeitsgraphen untersucht werden, die nur schon als fehlerhaft erkannte Varianten repräsentieren. Diese Optimierung wird durch die roten Rechtecke in Algorithmus 1 gezeigt. Sobald ein Zustand als fehlerhaft erkannt wird, werden die durch ihn repräsentierten Systemvarianten von der weiteren Suche ausgeschlossen. Dieses Verhalten ermöglicht einen direkten Vergleich mit variantenbasierten Modelchecking, bei dem auch höchstens ein Fehler pro Variante gefunden werden kann.

Evaluierung Um die verschiedenen Analysestrategien empirisch zu vergleichen, haben wir neun konfigurierbare Systeme ausgewählt und jeweils mit den varianten-, familien- und stichprobenbasierten Strategien analysiert. Zur Analyse haben wir die Modelchecker Java Pathfinder (für Java) und CPAchecker (für C) verwendet. Zur Evaluierung der familienbasierten Strategie haben wir die Werkzeuge dabei mit den oben beschriebenen Optimierungen erweitert.

Wir haben die stichprobenbasierte Strategie in drei Ausprägungen untersucht: single-wise, pair-wise und triple-wise. Die Ausprägungen unterscheiden sich darin, welche Kombinationen von Konfigurationsoptionen in der Stichprobe abgedeckt werden. Bei der pair-wise Stichproben-Strategie werden z.B. alle Paare von Konfigurationsoptionen in mindestens einer Variante in der Stichprobe zusammen ausgewählt.

Abbildung 4 zeigt ein Ergebnis dieser Evaluierung. Die Balken geben jeweils an, wie lange die Analyse aller Systemvarianten mit der jeweiligen Strategie dauert (bei stichprobenbasierter Strategie: alle Varianten in der Stichprobe). Die Abbildung zeigt, dass die familienbasierte Strategie immer schneller zum Ergebnis kommt als

die variantenbasierte Strategie. Weiterhin sind einige stichprobenbasierte Analysen manchmal schneller als die familienbasierte Strategie. Man muss hier jedoch berücksichtigen, dass die stichprobenbasierten Analysen oft nur sehr wenige Systemvarianten analysieren und damit viel weniger aussagekräftige Ergebnisse erzeugen.

4 Weitere Themen und Fazit

Konfigurierbare Systeme haben oft eine riesige Anzahl von Varianten, die die variantenbasierte Analyse sehr teuer macht. Das Ziel dieser Arbeit ist, die effiziente Analyse hochkonfigurierbarer Softwaresysteme zu ermöglichen. Wir haben uns mit der Frage beschäftigt, wie Programmanalysen (z.B. Datenflussanalyse und Modelchecking) auf konfigurierbare Softwaresysteme angewendet werden können. Zu diesem Zweck entwickelten wir das PLA-Modell zur Beschreibung von Analysestrategien für konfigurierbare Systeme (nicht in dieser Zusammenfassung beschrieben). Das PLA-Modell umfasst die vier grundlegenden Analysestrategien (varianten-, stichproben-, feature- und familienbasiert) und verschiedene Kombinationen dieser Strategien. Basierend auf dem Modell diskutierten wir Vor- und Nachteile verschiedener Kombinationen von Basisstrategien. Darüber hinaus haben wir mehrere Analysestrategien implementiert und sie evaluiert. Unsere Evaluierungen haben gezeigt, dass abhängig von dem analysierten konfigurierbaren System und der Art der Analyse (z.B. Typprüfung oder Modelchecking) die Auswahl einer guten Analysestrategie entscheidend für die Effizienz ist. Die Ergebnisse der familien- und variantenbasierten Strategien decken alle Varianten ab; die stichprobenbasierten Strategie analysiert nur einen Teil. Darüber hinaus ist die familienbasierte Strategie oft schneller als die stichproben- und variantenbasierten Strategien. Die familienbasierte Strategie kann jedoch mehr Hauptspeicher verbrauchen andere Strategien.

Eine familienbasierte Analyse, die durch eine variabilitätsgewahre Datenstruktur unterstützt wird, ermöglicht die Analyse von sehr großen Systemen. Wir haben diese Kombination in einer Taint-Propagation-Analyse der Kommunikation zwischen Apps eines Android App Stores verwendet. Durch die Kombination der familien- und variantenbasierten Strategien lässt sich die Analyseleistung verbessern. Diese Analyse und ihre Evaluierung werden in dieser Zusammenfassung nicht weiter diskutiert.

Die Techniken, die in dieser Dissertation entwickelt wurden, insbesondere die Vereinfachung von Präsenzbedingungen und die Variabilitätskodierung, haben Auswirkungen über unsere eigene Forschung hinaus. Sie werden in vielen familienbasierten Analysen verwendet oder ermöglichen diese erst [Ap13; SRA13]. Die Auswirkungen dieser Arbeit auf die Praxis werden vor allem durch unsere Werkzeuge (Hercules, SPLverifier und SIFTA) und unsere Auswertungen dargestellt. Diese Tools und Evaluierungen helfen Entwicklern von Analysen für konfigurierbare Systeme entweder direkt oder bei der Wahl der Analysestrategie.

Literaturverzeichnis

[Ap11] Apel, S.; Speidel, H.; Wendler, P.; von Rhein, A.; Beyer, D.: Detection of Feature Interactions using Feature-Aware Verification. In: Proceedings of the International Conference on Automated Software Engineering (ASE). IEEE, S. 372–375, 2011.

[Ap13] Apel, S.; von Rhein, A.; Wendler, P.; Größlinger, A.; Beyer, D.: Strategies for Product-Line Verification: Case Studies and Experiments. In: Proceedings of the International Conference on Software Engineering (ICSE). IEEE, S. 482–491, 2013.

[Be15] Beyer, D.: Software Verification and Verifiable Witnesses (Report on SV-COMP 2015). In: Proceedings of the International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS). Springer-Verlag, S. 401–416, 2015.

[Cl11] Classen, A.; Heymans, P.; Schobbens, P.-Y.; Legay, A.: Symbolic Model Checking of Software Product Lines. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, S. 321–330, 2011.

[NL11] Nie, C.; Leung, H.: A Survey of Combinatorial Testing. ACM Computing Surveys 43/2, 11:1–11:29, 2011.

[OMR10] Oster, S.; Markert, F.; Ritter, P.: Automated Incremental Pairwise Testing of Software Product Lines. In: Proceedings of the International Software Product Line Conference (SPLC). Springer-Verlag, S. 196–210, 2010.

[PS08] Post, H.; Sinz, C.: Configuration Lifting: Verification meets Software Configuration. In: Proceedings of the International Conference on Automated Software Engineering (ASE). IEEE, S. 347–350, 2008.

[Rh16] von Rhein, A.: Analysis Strategies for Configurable Systems, Diss., University of Passau, Germany, 2016.

[SRA13] Siegmund, N.; von Rhein, A.; Apel, S.: Family-Based Performance Measurement. In: Proceedings of the International Conference on Generative Programming: Concepts & Experience (GPCE). ACM, S. 95–104, 2013.

Alexander von Rhein studierte von 2005 bis 2010 Informatik an der Universität Passau. Dort folgte im Juni 2016 seine Promotion (Prädikat "summa cum laude") am Lehrstuhl von Prof. Dr.-Ing. Sven Apel. Sein Forschungsinteresse gilt Softwareproduktlinien, automatisierter Verifikation und Analyse von Software sowie formalen und empirischen Methoden. Seit seiner Promotion arbeitet Alexander von Rhein als Berater für Software Qualität bei der CQSE GmbH in Garching. Er widmet sich der Entwicklung von Analysewerkzeugen mit speziellen Kundenanforderungen und dem Transfer von Forschungsergebnissen in die industrielle Praxis.