Brückenschlag zwischen Verifikation und systematischem Testen[1]

Maria Christakis[2]

1 Einführung

Die Forschung beschäftigt sich seit circa fünf Jahrzehnten mit Verifikation und sie wird vermehrt in der Industrie dazu eingesetzt, um während der Softwareentwicklung Fehler zu erkennen. Verifikationstechniken basieren darauf, dass Softwareentwickler die von ihnen gewünschten Eigenschaften des Programms vorgeben. Anschliessend können verschiedene Techniken dazu eingesetzt werden, um diese Korrektheitseigenschaften zu beweisen. Inzwischen sind Verifikationswerkzeuge so erfolgreich, dass sie vermehrt routinemassig von vielen Softwareentwicklern dazu eingesetzt werden, um Fehler in industriell eingesetzter Software zu finden. In der Tat gibt es inzwischen eine Vielzahl solcher Werkzeuge für etablierte Programmiersprachen. Diese decken ein sehr breites Spektrum ab: einfache heuristische Werkzeuge, Werkzeuge die auf Abstrakter Interpretation basieren, Software-Model-Checker, bis hin zu Verifikationswerkzeugen, die auf automatischen Theorembeweisern aufbauen.

Im Laufe der letzten zehn Jahre keimte gleichzeitig ein vermehrtes Interesse an systematischem Testen auf. Die meisten dieser Techniken basieren auf symbolischer Programmausführung, welche vor mehr als drei Jahrzehnten entwickelt wurde. Die Entwicklung von dynamischem symbolischen Testen wurde durch erhebliche Fortschritte im Bereich der Constraint Satisfiability und der zunehmenden Skalierbarkeit von simultaner konkreter und symbolischer Programmausführung begünstigt. Diese Techniken erlauben es eine hohe Testabdeckung des Programms zu erreichen und tiefliegende Fehler in grossen und komplexen Programmen auszumachen. Dynamisches symbolisches Testen ist daher von grosser Bedeutung für viele Forschungsbereiche in der Informatik, wie zum Beispiel Software Engineering, Sicherheit, Computersysteme, Debugging und Fehlerbehebung, Netzwerke, Bildung und viele mehr.

Der Fokus dieser Dissertation [Ch15] liegt darauf die Kluft zwischen diesen beiden etablierten Techniken—Verifikation auf der einen Seite und systematisches Testen auf der anderen Seite—zu verringern. Auf der einen Seite ergänzen wir Verifikation mit systematischem Testen, um die Software-Qualität zu maximieren und indem wir gleichzeitig den Testaufwand reduzieren. Im Besonderen definie-

[1] Narrowing the gap between verification and systematic testing

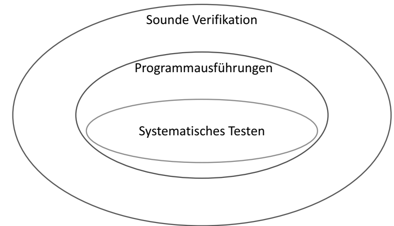
[2] Microsoft Research, Redmond, USA, mchri@microsoft.com

ren wir präzise welche Korrektheitsgarantien Verifikationswerkzeuge bieten. Erst dies erlaubt uns solche Werkzeuge wirkungsvoll mit dynamischem symbolischem Testen zu ergänzen. Gleichzeitig erweitern wir systematische Test-Techniken mit besseren Test-Orakeln, die es uns erlauben, bisher ignorierte Aspekte der Programmausführung in Betracht zu ziehen und dadurch mehr Fehler mit weniger Ressourcen und früher im Entwicklungsprozess auszumachen. Auf der anderen Seite untersuchen wir in wie weit systematisches Testen zur Verifikation von realistischen Applikationen benutzt werden kann. Im Besonderen untersuchen wir im Rahmen einer bestimmten Kategorie von Applikationen, ob systematisches Testen in der Lage ist Verifikation zu erreichen. Diese Forschungsrichtung lotet das Potenzial von dynamischem systematischem Testen zur Sicherstellung von Software-Korrektheit aus.

Im Folgenden erläutern wir die Motivation und Idee hinter den beiden Forschungsrichtungen dieser Dissertation: die Kombination von Verifikation und systematischem Testen und Verifikation mittels systematischem Testen.

2 Kombination von Verifikation und Systematischem Testen

Typischerweise betrachten sounde Verifikationstechniken eine Überapproximation der möglichen Programmausführungen, um die Abwesenheit von Fehlern in einem Programm zu beweisen. Im Gegensatz dazu versucht systematisches Testen die Existenz von Fehlern zu beweisen, indem eine Unterapproximation der möglichen Programmausführungen betrachtet wird.



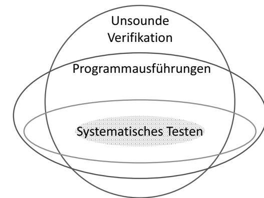
In der Praxis setzen Software-Projekte heutzutage auf eine Vielzahl von Techniken, wie zum Beispiel Testen, Code Reviews und statische Analyse, um Fehler in Programmen zu finden. In der Regel betrachten jedoch all diese Techniken nicht alle möglichen Programmausführungen. Oft werden ganze Programmpfade nicht verifiziert (beispielsweise wenn eine Test-Suite nicht vollständige Pfadabdeckung erreicht), einige Eigenschaften werden nicht verifiziert (beispielsweise komplexe Assertions), oder einige Pfade werden lediglich under Annahmen (beispielsweise dass keine arithmetischen Überläufe auftreten) verifiziert, die nicht zwangsläufig in allen Programmausführungen halten. In Code Reviews sind solche Annahmen nötig, um die Komplexität für den Gutachter in Grenzen zu halten. Ferner ist es üblich in statischen Analysen Annahmen zu treffen, um die Präzision, Effizienz und Modularität der Analyse zu erhöhen [CMW15] und da sich gewisse Programmfeatures einer statischen Überprüfung entziehen. In anderen Worten verzichten die meisten statischen Analysen zu Gunsten von anderen wichtige Qualitäten auf Soundness.

Obwohl statische Analysen wirkungsvoll zum Aufspüren von Software-Fehlern verwendet werden können, sind sie nicht in der Lage den Testaufwand zu ersetzen

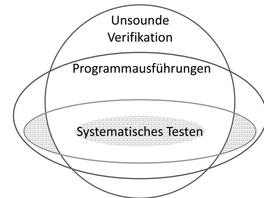
oder erheblich zu reduzieren. Viele solcher Analysen für etablierte Programmiersprachen machen verschiedene Kompromisse, um die Automatisierung zu steigern und den manuellen Annotationsaufwand, die Anzahl der unechten Fehler und die Dauer der Analyse zu reduzieren. Solche Kompromisse bestehen darin, dass gewisse Eigenschaften (beispielsweise Terminierung) nicht überprüft werden, implizite Annahmen getroffen werden (beispielsweise dass arithmetische Operationen nicht überlaufen) und Unsoundness (beispielsweise dass nur eine beschränkte Anzahl von Schleifeniterationen betrachtet werden).

Trotz dieser Limitationen finden solche statischen Analysen echte Fehler in industrieller Software. Aufgrund dieser Limitationen ist es jedoch nicht klar, welche Korrektheitsgarantien solche statischen Analysen besitzen. Es ist ebenfalls unklar, wie sich eben diese nicht sound verifizierten Eigenschaften mittels systematischem Testen überprüfen lassen. Daher sind Software-Ingenieure gezwungen Programme so gründlich zu testen, als wenn keine statischen Analysen zum Einsatz gekommen wären. Dies ist ineffizient, da es eine grosse Test-Suite voraussetzt.

Bis heute wurden mehrere Ansätze entwickelt, um Verifikation mit Testen zu ergänzen. Diese dienen jedoch mehrheitlich dazu festzustellen, ob es sich bei einem statisch entdeckten Fehler um einen echten Fehler handelt. Keiner dieser Ansätze zieht jedoch in Betracht, dass statische Analysen Fehler übersehen können aufgrund der bereits erwähnten Limitationen. In anderen Worten zielt das Testen lediglich darauf ab, Programmausführungen zu überprüfen, für welche die Verifikation fehlschlug. Wie in der Abbildung rechts ersichtlich wird, werden dabei Programmausführungen ignoriert, die aufgrund von Unsoundness nicht statisch überprüft wurden.



Um dieses Problem in den Griff zu bekommen, haben wir eine neue Technik zur Kombination von Verifikation und systematischem Testen entwickelt. Diese Technik leitet das systematische Testen nicht nur zu den Programmausführungen, für welche die Verifikation fehlschlug, sondern auch zu jenen, die aufgrund von Unsoundness nicht verifiziert wurden. Die schattierten Flächen in der Abbildung rechts zeigen ebendiese Programmausführungen.



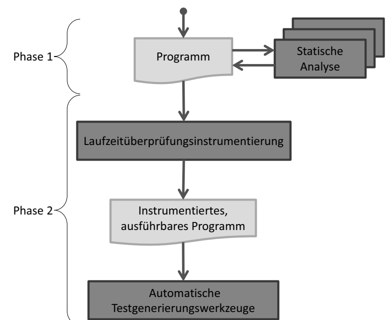
Im Besonderen präsentieren wir eine Werkzeug-Architektur die (1) mehrere, komplementäre statische Analysen, welche unterschiedliche Eigenschaften überprüfen und unterschiedliche Annahmen treffen, mit (2) dynamischem systematischem Testen ergänzt, um Eigenschaften zu überprüfen, die statisch nicht bereits überprüft wurden [CMW12]. Der Schlüsselgedanke hinter dieser Architektur besteht darin, explizit zu machen, welche Eigenschaften statisch überprüft wurden und unter wel-

chen Annahmen. Dies erlaubt es, die Korrektheitsgarantien einer statischen Analyse präzise zu dokumentieren und ermöglicht bei der Test-Generierung jene Eigenschaften ins Auge zu fassen, die noch nicht verifiziert wurden. Das Resultat davon sind kleinere und wirkungsvollere Test-Suiten.

Die drei wissenschaftlichen Errungenschaften unser Architektur lassen sich folgendermassen zusammenfassen:

1. Sie macht absichtliche Kompromisse von statischen Analysen explizit, indem jede Assertion entweder als vollständig verifiziert, unter Annahmen verifiziert oder nicht verifiziert gekennzeichnet wird.
2. Sie generiert Testfälle automatisch basierend auf den Resultaten der statischen Analysen. Dies erlaubt es dem Benutzer zu entscheiden wie viel Aufwand er in die statischen Analysen und das Testen stecken möchte. Beispielsweise hat der Benutzer die Wahl keinerlei Aufwand zu betreiben, um die Verifikation vollständig durchzuführen (z.B. indem er keinerlei Invarianten für Schleifen schreibt). Das Verifikationswerkzeug ist in der Lage einige Eigenschaften zu beweisen und unsere Architektur erlaubt es, die übrigen zu testen. Wahlweise hat der Benutzer die Möglichkeit möglichst viele Eigenschaften von kritischen Software-Komponenten zu beweisen und die übrigen Eigenschaften (z.B. in Bibliothekskomponenten) zu testen. Somit ist das Mass an statischer Analyse konfigurierbar und kann von nichts bis vollständig reichen.
3. Sie richtet die statische Analyse und das Testen auf die Eigenschaften die noch nicht (sound) überprüft wurden. Dies erlaubt gezielte statische Analyse und Testen und, im speziellen, führt zu kleineren und wirkungsvolleren Test-Suiten.

Unsere Architektur nimmt als Eingabe ein Programm, bestehend aus Code, Spezifikationen und den Eigenschaften, welche überprüft werden sollen (z.B. Division-durch-Null und Null-Dereferenzierung). Für jede Überprüfung in dem Programm hält unsere Architektur fest, ob sie sound verifiziert wurde oder unter Annahmen. Wie in der Abbildung rechts ersichtlich wird, besteht die Architektur aus zwei Phasen, wobei die erste aus statischen Analysen besteht und die zweite aus systematischem Testen.



Die erste Phase erlaubt dem Benutzer eine beliebige Anzahl (eventuell keine) von statischen Analysen durchzuführen. Jede Analyse liest das Programm, welches Resultate aus früheren Analysen enthalten kann, und versucht Eigenschaften zu beweisen, die noch nicht von vorgeschalteten Analyse-Werkzeugen bewiesen wurden. Wie bereits beschrieben, ist jede Assertion entweder als vollständig (d.h. sound) verifiziert, unter gewissen Annahmen verifiziert oder nicht verifiziert (d.h. Verifikation wurde nicht versucht oder ist fehlge-

schlagen) gekennzeichnet. Eine Analyse versucht dann diejenigen Assertions zu beweisen, die noch nicht von vorgeschalteten Analyse-Werkzeugen bewiesen wurden. Jede Analyse erfasst ihre Resultate im Programm, welches wiederum als Eingabe für das nachgeschaltete Werkzeug dient.

Das Programm, das von der ersten Phase produziert wird, ist entweder vollständig (und sound) verifiziert oder es existieren noch Überprüfungen, die nicht bewiesen werden konnten. Jede Zwischenversion des Programms erfasst genau, welche Eigenschaften bereits vollständig verifiziert wurden und welche noch zu validieren sind. Dies macht es den Entwicklern leicht den Verifikationszyklus jederzeit zu unterbrechen, was in der Praxis entscheidend ist, da Entwickler nur einen begrenzten Aufwand für die statische Analysen betreiben können. Die übrigen Eigenschaften können durch die anschliessende Test-Phase abgedeckt werden.

In dieser zweiten Phase setzen wir dynamisches symbolisches Testen ein, um automatisch Testfälle anhand des (in der Regel bereits statisch analysierten) Programms zu generieren. Insbesondere nutzen wir eine instrumentierte Version des Programms, in der sowohl Eigenschaften, die noch zu überprüfen sind, als auch die Annahmen der statischen Analysen als Laufzeit-Überprüfungen vorkommen. Das resultierende instrumentierte Programm kann dann mit einem oder mehreren Testgenerierungswerkzeugen getestet werden. Unsere Instrumentierung führt dazu, dass das symbolische Testen genau die Constraints und Testdaten generiert, um die Eigenschaften zu überprüfen, die noch nicht (sound) statisch verifiziert wurden. Dies reduziert die Anzahl generierter Tests. Es können jedoch nicht alle Spezifikationen effizient zur Laufzeit überprüft werden (beispielsweise Objektinvarianten) und Programme interagieren mit deren Umgebung auf verschiedene Arten (beispielsweise über den statischen Zustand). Daher benutzen wir in dieser Phase Strategien, um die Resultate des systematisch Testen bezüglich der wirklich überprüften Eigenschaften und der entdeckten Fehler zu verbessern.

Indem wir diese Architektur entwickelten, waren wir in der Lage die folgenden wissenschaftlichen Themen zu untersuchen.

Wie entwirft man eine Annotationssprache die Verifikation und systematisches Testen unterstützt. Seit Design by Contract wurden mehrere verwandte Annotationssprachen entwickelt. Beispielsweise wurden Eiffel, JML (Java Modeling Language), Spec# und Code Contracts für .NET so entworfen, dass sie sich sowohl für statische Analysen als auch für die Laufzeitüberprüfung von Annotationen eignen. Keine dieser Sprachen eignet sich jedoch, um unser Architektur für Werkzeuge zu verwirklichen.

Wir haben daher eine Annotationssprache entwickelt, um absichtliche Kompromisse von statischen Analysen explizit zu machen [CMW12] (siehe Kapitel 2). Die Haupttugenden unserer Annotationen sind, (1) dass sie leicht von einer Vielzahl von statischen und dynamischem Werkzeugen verstanden werden können [CMW16], (2) dass sie ausdrucksstark sind, wie wir nachgewiesen haben, indem wir typische Kompromisse von deduktiven Verifikationswerkzeugen [CMW12] und des abstrak-

ten Interpreters Clousot für .NET ausgedrückt haben und (3) dass sie sich gut für Testgenerierung eignen [CMW12, CMW16].

Welches sind die Limitationen von etablierten Verifikationswerkzeugen und wie lassen sie sich explizit machen. Viele etablierte statische Analysen machen Kompromisse, um die Automatisierung oder den Durchsatz zu verbessern oder um die Anzahl der unechten Fehler oder den Annotationsaufwand zu reduzieren. Beispielsweise benutzt HAVOC Schreibeffektspezifikationen ohne diese zu überprüfen, Spec# ignoriert arithmetische Überläufe und Kontrollfluss für Exceptions, ESC/Java betrachtet nur eine beschränkte Anzahl Schleifeniterationen, der abstrakte Interpreter Clousot für .NET benutzt eine unsound Heap-Abstraktion, KeY unterstützt Multi-Objektinvarianten nicht auf sounde Art und Weise, Krakatoa handhabt Klasseninvarianten und Klasseninitialisierung nicht auf sounde Art und Weise und Frama-C benutzt Plug-ins für verschiedene Analysen, selbst wenn sie widersprüchliche Annahmen treffen.

Solange man die Kompromisse dieser Analysen explizit macht, können ihre Benutzer unmittelbar von unserer Architektur profitieren, welche die Zusammenarbeit von Analysen unterstützt und wirkungsvoll von Testgenerierungswerkzeugen ergänzt werden kann.

Wir haben unsere Annotationen benutzt, um typische Kompromisse von deduktiven Verifikationswerkzeugen explizit zu machen [CMW12]. Gleichzeitig haben wir die meisten Soundness-Kompromisse in Clousot, einem weitverbreiteten kommerziellen statischen Analysewerkzeug, ausgedrückt. Wir haben die Auswirkungen der unsounden Annahmen in Clousot für mehrere Open-Source-Projekte gemessen und damit die erste systematische Anstrengung unternommen, um die Quellen der Unsoundness in einem praktischen Analysewerkzeug zu dokumentieren und zu untersuchen [CMW15].

Wie ergänzt man Verifikation mit systematischem Testen, sodass Code-Qualität maximiert wird und Testaufwand minimiert wird. Wie bereits erwähnt, existieren mehrere Ansätze, um unsounde Verifikation mit Testen zu ergänzen, hauptsächlich jedoch um festzustellen, ob ein Verifikationsfehler unecht ist. Unsere Architektur ist ebenfalls in der Lage, zu bestätigen, ob ein Verifikationsfehler auch tatsächlich ein echter Fehler ist. Die Phase, welche die Laufzeitüberprüfungen hinzufügt, führt Assertions für all jene Eigenschaften ein, die noch nicht statisch verifiziert wurden (inklusive fehlgeschlagener Verifikationsversuche). Die Test-Phase benutzt nun diese Assertions, um die Testgenerierungswerkzeuge zu den Eigenschaften zu leiten, welche noch nicht verifiziert wurden. Schlussendlich generieren die Testgenerierungswerkzeuge entweder eine Reihe von erfolgreichen Testfällen, welche das Vertrauen der Benutzer in die Korrektheit des Programms steigern, oder ein konkretes Gegenbeispiel, das einen Fehler reproduziert.

Wir haben unsere Toolchain für das statische .Net-Analysewerkzeug Clousot und das Testgenerierungswerkzeug Pex entwickelt, um zu untersuchen, wie sich unso-

unde statische Analyse mit systematischem Testen mittels unserer Annotationen ergänzen lässt. In diesem Rahmen haben wir untersucht, wie sich diese Annotationen optimal ausnutzen lassen, um die Testgenerierung zu den Eigenschaften zu führen, welche noch nicht sound von der statischen Analyse überprüft wurden.

In Kapitel 2 präsentieren wir eine Technik, um die Redundanz mit der statischen Analyse zu reduzieren, wenn partielle Verifikationsresultate (ausgedrückt mithilfe unserer Annotationen) durch automatische Testgenerierung [CMW16] ergänzt werden. Unsere Haupterrungenschaft besteht aus einer Code-Instrumentierung, welche das dynamische systematische Testen dazu bringt, Testfälle vorzeitig zu beenden, welche zu verifizierten Programmausführungen führen, Teile des Suchraums zu eliminieren und Testfälle zu priorisieren, welche zu unverifizierten Programmausführungen führen. Diese Instrumentierung basiert auf einer effizienten statischen Inferenz, die Information über unverifizierte Programmausführungen im Kontrollfluss nach oben propagiert, wo sich der Suchraum wirkungsvoller eingrenzen lässt. Unsere Instrumentierung erlaubt es Pex kleinere Test-Suiten (bis zu 19.2%) zu produzieren, mehr unverifizierte Programmausführungen abzudecken (bis zu 7.1%) und die Test-Dauer zu reduzieren (bis zu 52.4%).

Es ist ebenfalls nützlich sounde, interaktive Verifikation mit systematischem Testen von Eigenschaften zu ergänzen, die noch nicht verifiziert wurden. In Kapitel 5, stellen wir Delfy vor, ein dynamisches Testgenerierungswerkzeug mit dem Ziel das sounde interaktive Verifikationswerkzeuge Dafny zu ergänzen. In diesem Kapitel untersuchen wir, wie sich interessante Sprach- und Spezifikationskonstrukte der Dafny Programmiersprache testen lassen. Wir erläutern ausserdem, wie man durch Testen vermeiden kann, dass man unnützerweise Zeit damit verbringt, ein inkorrektes Programm zu verifizieren, wie sich unechte Verifikationsfehler debuggen lassen und wie sich redundante Spezifikationen vermeiden lassen [Ch16].

Wie generiert man Testfälle für Eigenschaften, die schwer zu verifizieren sind und welche die Leistungsfähigkeit von systematischen Testwerkzeugen überschreiten. In der zweiten Phase unserer Architektur versuchen wir jene Eigenschaften zu testen, welche noch nicht sound von vorgeschalteten statischen Analysen verifiziert wurden. Da unser Endziel darin besteht automatisch Indizien für die Korrektheit des Programms zu finden, muss diese Phase dazu in der Lage sein, sowohl wirkungsvolle Test-Orakel für unverifizierte, komplexe Eigenschaften (als Laufzeitüberprüfungen) zu erstellen, als auch Test-Inputs zu finden, welche diese Orakel gründlich validieren. Wir haben untersucht, wie sich ein attraktiver Kompromiss zwischen Durchsatz und Abdeckung von Orakeln erreichen lässt, indem eine einfache statische Analyse verwendet wird, um sowohl die Anzahl Orakel, welche noch überprüft werden müssen, als auch die Anzahl Test-Inputs, die jene Orakel beeinträchtigen, zu reduzieren.

In Kapitel 3 befassen wir uns mit einer Limitation von existierenden Testgenerierungswerkzeugen bei der Erstellung von Orakeln, die stark genug sind für komplexe Spezifikation. Insbesondere verlangen automatische Testgenerierungswerkzeuge eine Beschreibung der Inputdaten, welche das Unit-Under-Test handhaben soll. Im

Fall von Heap-Datenstrukturen wird eine solche Beschreibung typischerweise mittels einer Objektinvariante ausgedrückt. Wenn ein Programm jedoch Datenstrukturen erstellen kann, welche die Invariante verletzen, werden Testdaten systematisch nicht berücksichtigt, sofern sie mit Hilfe der Invariante erstellt wurden. Dies kann dazu führen, dass Fehler nicht erkannt werden. Wir lösen dieses Problem mit einer Technik, die Verletzungen von Objektinvarianten erkennt [CMW14]. Wir erläutern drei Szenarien, in welchen die traditionelle Art der Invariantenüberprüfung (wie für gewöhnlich in existierenden Testgenerierungswerkzeugen implementiert) solche Verletzungen übersehen kann. Basierend auf einer Reihe von vordefinierten Mustern, welche diese Szenarien abdecken, synthetisieren wird Parameterized-Unit-Tests, die voraussichtlich die Invarianten verletzen, und verwenden dynamisches systematisches Testen, um Inputs für die synthetisierten Tests zu erstellen. Wir haben diese Technik als eine Erweiterung des dynamischen Testgenerierungswerkzeugs Pex implementiert und haben sie auf Open-Source-Projekte angewendet, welche sowohl manuell von Programmierern erstellte Invarianten enthalten als auch Invarianten, die von Daikon automatisch inferiert wurden. In beiden Fällen konnten wir eine erhebliche Anzahl von Invariantenverletzungen ausmachen.

In Kapitel 4 betrachten wir eine zweite Limitation von existierenden Testgenerierungswerkzeugen bei der Erstellung von geeigneten Inputs für ein Unit-Under-Test. Obwohl statischer Zustand häufig in objektorientierten Programmen verwendet wird, ignorieren automatische Testgenerierungswerkzeuge potentielle Interferenzen zwischen dem statischen Zustand und dem Unit-Under-Test, was dazu führen kann, dass subtile Fehler übersehen werden. Insbesondere betrachten existierende Testgenerierungswerkzeuge statische Felder nicht als Input des Unit-Under-Test und steuern die Ausführung von statischen Initialisierungen nicht. Wir lösen dieses Problem mittels einer neuartigen Technik zur automatische Testgenerierung [CEM14], welche auf statischer Analyse und dynamischem systematischem Testen basiert. Wir haben unsere Technik auf eine Reihe von Open-Source-Projekte angewendet und Fehler ausgemacht, die von existierenden Testgenerierungswerkzeugen nicht erkannt werden. Unsere Experimente zeigen, dass dieses Problem tatsächlich in der Realität auftritt, sie geben Hinweise darauf welche Arten von Fehlern nicht von existierenden Werkzeugen entdeckt werden und sie demonstrieren die Wirksamkeit unserer Technik.

3 Verifikation mittels Systematischem Testen

Im Laufe der letzten Dekade wurde dynamisches systematisches Testen in einer Vielzahl von weitverbreiteten Werkzeugen implementiert. Einige Beispiele sind EXE, jCUTE, Pex, KLEE, BitBlaze, und Apollo. Obwohl diese Werkzeuge wirkungsvoll dazu eingesetzt werden Fehler zu finden, wurden sie nie zur Verifikation von grossen und komplexen Programmen verwendet; d.h um zu Beweisen, dass in dem Programm keine Fehler einer gewissen Art auftreten. Im zweiten Teil dieser Dissertation untersuchen wir, inwieweit Verifikation mit systematischem Testen in der Praxis erreicht werden kann. Dazu beschränken wir uns auf eine bestimmte

Kategorie von Applikationen und zwar von Binary-Image-Parsern. Insbesondere untersuchen wir, im Rahmen dieser Kategorie von Applikationen, ob es realistisch ist, dass alle möglichen Programmausführungen mittels systematischem Testen abgedeckt werden können.

In Kapitel 6 erläutern wir unsere Erweiterung von dynamischem systematischem Testen zur Verifikation des ANI-Image-Parsers in Windows, welcher in systemnahem C geschrieben ist [CG15b]. Um dies zu erreichen, haben wir lediglich drei Kerntechniken angewendet, nämlich (1) symbolische Programmausführung auf der Stufe des x86-Binärprogramms, (2) erschöpfende Aufzählung und Testen aller Programmpfade und (3) Programmdekomposition und Summarization mit Hilfe des Benutzers. Wir haben SAGE und ein neuartiges Werkzeug namens MicroX verwendet, um Teile des Programms isoliert mittels einer speziell fürs Testen entwickelten virtuellen Maschine auszuführen. Dies erlaubt es uns erstmalig zu beweisen, dass ein komplexer Image-Parser in Windows memory-safe ist; d.h. frei von jeglichen Buffer-Overflow-Anfälligkeiten, vorausgesetzt dass unsere Werkzeuge sound sind und dass weitere Annahmen zutreffen bezüglich der Beschränkung von Schleifen, die vom Input abhängen, der Behebung von Buffer-Overflow-Fehlern und abgesehen von einigen Teilen des Programms, die absichtlich nicht memory-safe sind. Dieser Prozess führte ausserdem dazu, dass mehrere Limitationen in unseren Werkzeugen aufgedeckt und eliminiert wurden.

In Kapitel 6, beschränken wir die Pfadexplosion mittels Programmdekomposition und Summarization mit Hilfe des Benutzers. Insbesondere zerlegen wird das Programm lediglich entlang einiger weniger Funktionsinterfaces. Diese sind sehr einfach gestrickt, was es uns ermöglicht die Summaries effizient logisch auszudrücken. Basierend auf den dabei erzielten Erkenntnissen, erläutern wir IC-Cut (Interface-Complexity-based Cut) [CG15a], eine compositional Search-Stratey für systematisches Testen von grossen Programmen, in Kapitel 7. IC-Cut ermittelt auf dynamische Art und Weise Funktionsinterfaces, welche einfach genug sind, um dafür kostengünstig ein Summary zu erstellen. IC-Cut zerlegt dann das Programm hierarchisch in Teile, welche durch Funktionen und deren Unter-Funktionen im Call-Graph definiert sind. Diese Teile werden unabhängig voneinander getestet, wobei die Test-Resultate in Summaries von geringer Komplexität überführt werden, und diese Summaries werden anschliessend zum Testen der höheren Funktionsstufen im Call-Graph verwendet. Dies ermöglicht uns, die Pfadexplosion zu beschränken. Wenn die zerlegten Teile erschöpfend getestet wurden, entsprechen sie verifizierten Komponenten des Programms. IC-Cut wird dynamisch während der Suche ausgeführt, was typischerweise dazu führt, dass Cuts im Laufe der Suche verfeinert werden. Wir haben diesen Algorithmus als eine neue Search-Stratey in SAGE entwickelt und wir präsentieren detaillierte experimentelle Resultate, die beim Testen des ANI-Image-Parsers in Windows erzielt wurden. Unsere Resultate zeigen, dass IC-Cut, im Vergleich zur gegenwärtigen Generational-Search-Strategy in SAGE, die Pfadexplosion lindert, wohingegen die Codeabdeckung und die Fähigkeit Fehler zu finden erhalten oder sogar gesteigert wird.

Danksagung. Ich danke Valentin Wüstholz für die deutsche Übersetzung dieser Abhandlung und für seine Hilfe und sein Feedback.

Literaturverzeichnis

[CEM14] Christakis, Maria; Emmisberger, Patrick; Müller, Peter: Dynamic Test Generation with Static Fields and Initializers. In: RV. Jgg. 8734 in LNCS. Springer, S. 269–284, 2014.

[CG15a] Christakis, Maria; Godefroid, Patrice: IC-Cut: A Compositional Search Strategy for Dynamic Test Generation. In: SPIN. Jgg. 9232 in LNCS. Springer, S. 300–318, 2015.

[CG15b] Christakis, Maria; Godefroid, Patrice: Proving Memory Safety of the ANI Windows Image Parser Using Compositional Exhaustive Testing. In: VMCAI. Jgg. 8931 in LNCS. Springer, S. 373–392, 2015.

[Ch15] Christakis, Maria: Narrowing the Gap between Verification and Systematic Testing. Dissertation, ETH Zurich, 2015.

[Ch16] Christakis, Maria; Leino, K. Rustan M.; Müller, Peter; Wüstholz, Valentin: Integrated Environment for Diagnosing Verification Errors. In: TACAS. LNCS. Springer, 2016. To appear.

[CMW12] Christakis, Maria; Müller, Peter; Wüstholz, Valentin: Collaborative Verification and Testing with Explicit Assumptions. In: FM. Jgg. 7436 in LNCS. Springer, S. 132–146, 2012.

[CMW14] Christakis, Maria; Müller, Peter; Wüstholz, Valentin: Synthesizing Parameterized Unit Tests to Detect Object Invariant Violations. In: SEFM. Jgg. 8702 in LNCS. Springer, S. 65–80, 2014.

[CMW15] Christakis, Maria; Müller, Peter; Wüstholz, Valentin: An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzer. In: VMCAI. Jgg. 8931 in LNCS. Springer, S. 336–354, 2015.

[CMW16] Christakis, Maria; Müller, Peter; Wüstholz, Valentin: Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In: ICSE. ACM, 2016. To appear.

Maria Christakis ist zur Zeit ein Post-doctoral Researcher in den Research in Software Engineering (RiSE) und Tools for Software Engineers (TSE) Gruppen bei Microsoft Research in Redmond (USA). Sie schloss ihr Doktorat im Sommer 2015 am Departement für Informatik der ETH Zürich ab, wo sie das Glück hatte von Peter Müller betreut zu werden. Für ihre Dissertation erhielt sie die ETH-Medaille für hervorragende Dissertationen. Sie machte ihren Bachelor- und Master-Abschluss in Elektrotechnik und Informatik an der National Technical University of Athens (Griechenland).