Beiträge zu praktikabler Prädikatenanalyse[1]

Philipp Wendler[2]

Abstract: Der Stand der Forschung im Bereich der automatischen Software-Verifikation ist fragmentiert. Verschiedene Verfahren existieren nebeneinander in unterschiedlichen Darstellungen und mit wenig Bezug zueinander, aussagekräftige Vergleiche sind selten. Die Dissertation adressiert dieses Problem. Ein konfigurierbares und flexibles Rahmenwerk zur Vereinheitlichung solcher Verfahren wird entwickelt und mehrere vorhandene Verfahren werden in diesem Rahmenwerk ausgedrückt. Dies bringt neue Erkenntnisse über die Kernideen dieser Verfahren, ermöglicht experimentelle Studien in einer neuartigen Qualität, und erleichtert die Forschung an Kombinationen und Weiterentwicklungen dieser Verfahren. Die Implementierung dieses Rahmenwerks im erfolgreichen Verifizierer CPACHECKER wird in der bisher größten derartigen experimentellen Studie (120 verschiedene Konfigurationen, 671 280 Ausführungen) evaluiert. Hierzu wird ein Benchmarking-System präsentiert, das mit Hilfe moderner Technologien signifikante qualitative Messfehler existierender Systeme vermeidet.

1 Einführung

Software ist ein wichtiger Bestandteil unseres Lebens und steuert viele sicherheitskritische Systeme wie z. B. Kraftwerke, Flugzeuge, Züge und Autos. Nicht nur in solchen Systemen ist die Korrektheit der eingesetzten Software von höchster Wichtigkeit. Testen ist ein gängiges Verfahren zum Finden von Fehlern in Software, kann jedoch nicht alle Fehler finden bzw. deren Abwesenheit beweisen. Manuelle oder semi-automatische Verifikation ist aufgrund des Aufwands oft nicht einsetzbar. Automatische Verifikation versucht diese Lücke zu schließen und höhere Gewissheit als Testen mit weniger Aufwand als manuelle Verifikation zu erreichen: Gegeben ein Eingabeprogramm und eine Spezifikation soll ein Verifizierer automatisch entscheiden, ob das Programm die Spezifikation erfüllt oder nicht. Aufgrund der Unentscheidbarkeit des Problems kann es allerdings zu Fällen ohne nutzbarem Ergebnis kommen. Obwohl sich Verfahren wie Model-Checking z. B. in der Verifikation von Gerätetreibern in Betriebssystemen als nützlich erwiesen haben, werden sie von Software-Entwicklern allerdings nur selten eingesetzt.

Fortschritt im Bereich automatischer Software-Verifikation hängt ab von einem theoretischen Verständnis der Verfahren und der Möglichkeit, diese effektiv in der Praxis durch Benchmarking zu evaluieren. Hierbei können wir drei Probleme identifizieren:

1. Vorgestellte Verifikationsansätze unterscheiden sich oft grundlegend in ihrer Darstellung, selbst bei verwandten Ansätzen. Dies erschwert das Verständnis und die Möglichkeit, die Kernideen der verschiedenen Ansätze zu identifizieren und darauf aufbauend neue Ansätze zu entwickeln.

[1] Englischer Titel der Dissertation: „Towards Practical Predicate Analysis“

[2] Lehrstuhl für Software and Computational Systems, LMU München, philipp.wendler@lmu.de

2. Eine experimentelle Evaluation ist oft schwierig, da qualitativ hochwertige Implementierungen von Ansätzen nicht immer zur Verfügung stehen, und die vorhandenen Implementierungen in verschiedenen Verifizierern existieren, was aufgrund des Einflusses technischer Unterschiede die Vergleichbarkeit von Ergebnissen einschränkt.

3. Die zum Benchmarking im Rahmen von experimentellen Evaluationen eingesetzten Tools sind nicht immer zuverlässig und können Messfehler beliebiger Größe produzieren.

1.1 Zielsetzung

Ziel der Dissertation [We17] ist es, eine Lösung für diese Probleme zu finden. Wir fokussieren uns hierbei insbesondere auf Verfahren des Model-Checkings mit Hilfe von Prädikaten über Programmvariablen. Dies erlaubt es von der Mächtigkeit und Effizienz moderner Solver für Erfüllbarkeit modulo Theorien (SMT) zu profitieren. Bekannte prädikatenbasierte Verfahren sind z. B. Prädikatenabstraktion und Bounded Model-Checking, die in einer Reihe bekannter Verifizierer implementiert sind und in der Praxis eingesetzt werden.

In der Dissertation definieren wir ein flexibles theoretisches Rahmenwerk, in dem verschiedene bekannte Verfahren ausgedrückt und so vereinheitlicht werden. Dies ermöglicht es diese Verfahren zu vergleichen, ohne dass oberflächliche Darstellungsunterschiede dies erschweren, die jeweiligen Kernideen der Verfahren zu identifizieren, und neue, effektivere und effizientere, Kombinationen von Verfahren auf einfache Weise zu definieren. Außerdem entwickeln wir eine ausgereifte Implementierung dieses Rahmenwerks und damit aller darin ausgedrückten Verfahren im Verifizierer CPACHECKER [BK11]. Mit Hilfe dieser Implementierung evaluieren wir zum ersten Mal systematisch 120 verschiedene Konfigurationen von Ansätzen zur automatischen Software-Verifikation und gewinnen Erkenntnisse, die für Forscher und Nutzer gleichermaßen nützlich sind. Um die Zuverlässigkeit des dafür nötigen Benchmarkings zu gewährleisten, setzen wir moderne Technologien in einem neu entwickelten Benchmarking-Tool ein.

1.2 Replizierbarkeit der Forschungsergebnisse

Alle in der Dissertation vorgestellten Tools sind Open Source. Um die Replizierbarkeit der Ergebnisse sicherzustellen, stehen alle Eingabedaten der Experimente, die verwendeten Tools, sowie die Ergebnisse im Rohformat zum Download bereit [3].

1.3 Verwandte Arbeiten

Eine Vereinheitlichung von prädikatenbasierten Verfahren zur automatischen Software-Verifikation sowie detaillierte experimentelle und konzeptionelle Vergleiche dieser Ver-

[3] `https://www.sosy-lab.org/research/phd/wendler/`

fahren existierten bisher nicht. Vorreiter für eine solche Vereinheitlichung ist das CPA-Konzept [BHT07], das ein gemeinsames Rahmenwerk für statische Analysen und Model-Checking schafft, und hier verwendet wird. Vier Verifikationsverfahren werden in Abschnitt 3 vorgestellt, ein Überblick über weitere verwandte Verfahren und Verifizierer findet sich in der Dissertation [We17, Kapitel 7].

1.4 Hintergrund: Model-Checking

Model-Checking ist ein gängiges Verfahren zur Verifikation von Software. Hierbei wird ein abstraktes Modell des konkreten Programms erzeugt und für dieses geprüft, ob die Spezifikation erfüllt ist. Die Konstruktion des abstrakten Modells stellt sicher, dass falls das Modell die Spezifikation erfüllt, dies auch für das konkrete Programm gilt. Eine Spezifikationsverletzung im Modell impliziert jedoch nicht notwendigerweise eine solche im Programm (Überapproximation; solche Spezifikationsverletzungen werden „unecht“ genannt). Die Prüfung des abstrakten Modells kann z. B. durch eine Erreichbarkeitsanalyse auf abstrakten Zuständen erfolgen. Ein abstrakter Zustand repräsentiert eine Menge von konkreten Zuständen des Programms. Die Nachfolgerrelation des abstrakten Modells stellt sicher, dass die Nachfolger eines abstrakten Zustands *e* mindestens diejenigen konkreten Zustände abdecken, die das Programm ausgehend von den durch *e* repräsentierten konkreten Zuständen in einem Schritt erreichen kann. Durch Fixpunktiteration werden alle erreichbaren abstrakten Zustände aufgezählt und auf Spezifikationsverletzungen überprüft.

Die Art des abstrakten Modells und die Tatsache, ob es endlich ist, hängt von der Wahl der zugrunde liegenden abstrakten Domäne ab, die festlegt wie abstrakte Zustände repräsentiert werden. Gängige Beispiele sind Wertedomänen (abstrakte Zustände weisen einer Teilmenge von Programmvariablen je einen konkreten Wert zu), Intervaldomänen (abstrakte Zustände weisen jeder Programmvariablen Intervalle zu) und Prädikatendomänen (abstrakte Zustände bestehen aus einer booleschen Formel über Programmvariablen).

Das Abstraktionsniveau des abstrakten Modells ist entscheidend für den Erfolg der Verifikation: ein zu abstraktes Modell kann durch die Überapproximation zu unechten Gegenbeispielen und damit falschen Ergebnissen führen, ein zu konkretes Modell kann den Aufwand stark erhöhen. Um das passende Abstraktionsniveau zu finden, kann Gegenbeispiel-gesteuerte Abstraktionsverfeinerung (CEGAR) [Cl03] genutzt werden. Dabei wird mit einem sehr abstrakten Modell begonnen und bei jeder gefundenen Spezifikationsverletzung anhand des konkreten Programms überprüft, ob diese echt oder unecht ist. Bei einem unechten Gegenbeispiel wird das abstrakte Modell so weit wie nötig verfeinert (konkretisiert). Dieser Prozess liefert entweder eine echte Spezifikationsverletzung, einen Beweis, dass keine Spezifikationsverletzung im Programm existiert, oder terminiert nicht.

2 Eine flexible Domäne basierend auf Prädikaten

Wir definieren eine flexible und konfigurierbare abstrakte Domäne, die Programmzustände mit Hilfe von Prädikaten über Programmvariablen repräsentiert. Dabei drücken wir die

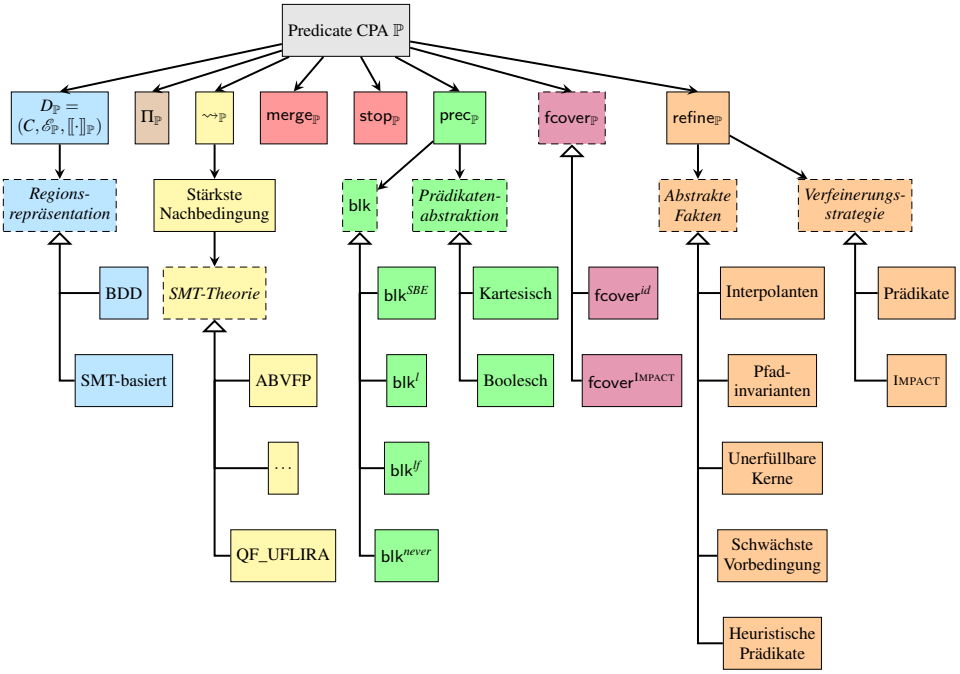

Abb. 1: Komponenten der Predicate CPA (aus [We17, BDW17])

abstrakte Domäne als konfigurierbare Programmanalyse (CPA) [BHT07] aus und nennen sie daher „*Predicate CPA*“. Die Predicate CPA besteht im Wesentlichen aus einem Halbverband $\mathscr{E}_{\mathbb{P}}$ von abstrakten Zuständen, einer Transferrelation $\rightsquigarrow_{\mathbb{P}}$ und den Operatoren $\mathsf{merge}_{\mathbb{P}}$, $\mathsf{stop}_{\mathbb{P}}$, $\mathsf{prec}_{\mathbb{P}}$, $\mathsf{fcover}_{\mathbb{P}}$ und $\mathsf{refine}_{\mathbb{P}}$. Einen Überblick über die verschiedenen Komponenten der Predicate CPA gibt Abb. 1. Um die Predicate CPA flexibel einsetzen zu können, sind bestimmte Komponenten austauschbar. In der Abbildung sind diese durch gestrichelte Boxen mit den darunterliegenden möglichen Varianten dargestellt. Im Folgenden werden beispielhaft einige wichtige Komponenten erläutert; eine vollständige Definition der Predicate CPA ist in der Dissertation [We17, Kapitel 4] enthalten.

Die abstrakten Zustände des Halbverbands $\mathscr{E}_{\mathbb{P}}$ bestehen aus booleschen Formeln über Prädikaten. Diese Formeln können entweder als binäres Entscheidungsdiagramm (BDD, aufwändig in der Erstellung, günstig in der Verwendung) oder als SMT-Formeln (günstig in der Erstellung, prüfen der Erfüllbarkeit ist teuer) repräsentiert werden. Die Transferrelation $\rightsquigarrow_{\mathbb{P}}$ ordnet einem abstrakten Zustand seine Nachfolgerzustände zu. Dies geschieht über die stärkste Nachbedingung und ohne die Berechnung einer Abstraktion. Je nach Einsatzzweck und verfügbaren Solvern können verschiedene Theorien zum Ausdrücken der stärksten Nachbedingung verwendet werden, von denen manche wie die Kombination von Feldern, Bitvektoren und Gleitkommazahlen es ermöglichen die Programmsemantik exakt abzubilden, während andere wie lineare Arithmetik nur eine Annäherung an die Programmsemantik abbilden können. Der Operator $\mathsf{prec}_{\mathbb{P}}$ kann einen abstrakten Zustand durch die Berechnung einer Prädikatenabstraktion [GS97] mit Hilfe eines SMT-Solvers

über eine gegebene Menge von Prädikaten weiter abstrahieren. Die Abstraktionsberechnung findet nur am Ende eines Blocks von Programmanweisungen statt, wobei die Blockgröße durch die Auswahl des Operators blk konfiguriert werden kann. Eine übliche Blockgröße ist z. B. die Wahl der größtmöglichen schleifenfreien Programmteile als Blöcke.

Ein weiterer wichtiger Operator ist der Operator $\mathsf{refine}_{\mathbb{P}}$, der den Verfeinerungsschritt des CEGAR-Ansatzes umsetzt. $\mathsf{refine}_{\mathbb{P}}$ prüft für jedes während der Analyse im abstrakten Modell gefundene Gegenbeispiel mit Hilfe eines SMT-Solvers, ob dieses Gegenbeispiel (d. h. ein Programmpfad zu einer Spezifikationsverletzung) im Programm valide ist. Falls dies zutrifft, ist eine Spezifikationsverletzung gefunden und die Analyse terminiert. Andernfalls ist das Gegenbeispiel unecht und die Genauigkeit der Analyse muss durch Verfeinerung des abstrakten Modells erhöht werden. Dazu berechnet $\mathsf{refine}_{\mathbb{P}}$ aus dem Gegenbeispiel passende abstrakte Fakten, die dem abstrakten Modell hinzugefügt werden müssen. Hierfür kann z. B. Craig-Interpolation [Cr57] genutzt werden. Anschließend wird das abstrakte Modell verfeinert, z. B. durch das Hinzufügen von Prädikaten aus den abstrakten Fakten zu jener Menge, die für die Abstraktionsberechnungen verwendet wird, oder durch das Stärken von abstrakten Zuständen durch Hinzukonjugieren der passenden abstrakten Fakten.

3 Vereinheitlichung von prädikatenbasierten Algorithmen

Aufbauend auf der Predicate CPA können wir nun ein Rahmenwerk definieren, das mehrere vorhandene prädikatenbasierte Algorithmen vereinheitlicht. Durch das Ausdrücken als CPA können wir den CPA-Algorithmus zur abstrakten Erreichbarkeitsanalyse [BHT07] mit nur wenigen Modifikationen verwenden [We17, BDW17]. Dieser Algorithmus berechnet die Menge aller erreichbaren abstrakten Zustände einer abstrakten Domäne wie der Predicate CPA durch eine Fixpunktiteration über die Transferrelation. Wir verwenden dabei die Predicate CPA in einer an das jeweils gewünschte Verfahren angepassten Konfiguration, d. h. wir wählen für jede der austauschbaren Komponenten (in Abb. 1 gestrichelt) eine bestimmte Variante aus. Außerdem setzen wir weitere auf dem CPA-Algorithmus aufbauende Algorithmen ein, die in der Dissertation detailliert beschrieben [We17, Kapitel 5] sind.

3.1 Bounded Model-Checking

Für Bounded Model-Checking [Bi99], d. h. das k-malige Abrollen des Programms und das Prüfen der Spezifikation für alle so erhaltenen endlichen Pfade, wird eine Konfiguration der Predicate CPA gewählt, die blk^{never} nutzt und ansonsten beliebig ist. Durch blk^{never} ist die Blockgröße unendlich groß und das gesamte Programm besteht aus einem einzigen Block. Darüber hinaus muss durch Kombination der Predicate CPA mit einer weiteren CPA, die die Beschränkung definiert (z. B. maximal k Schleifenabrollungen), sichergestellt werden, dass der CPA-Algorithmus terminiert. Anschließend wird durch einen einmaligen Aufruf eines SMT-Solvers überprüft, ob auf einem der gefundenen Pfade eine Spezifikationsverletzung erreichbar ist. Die dazu nötige Formel kann aus den abstrakten Zuständen abgelesen werden, die durch die unendliche Blockgröße nie abstrahiert wurden und daher die Programmsemantik (für den analysierten Teil) exakt abbilden.

3.2 k-Induktion

Mit Bounded Model-Checking können nur Spezifikationsverletzungen gefunden werden, jedoch im Allgemeinen keine Erfüllung der Spezifikation bewiesen werden. Dies lässt sich durch einen k-Induktionsbeweis [DKR11] durchführen. Hierbei besteht der Induktionsanfang aus der Überprüfung, dass in den ersten k Schleifeniterationen die Spezifikation erfüllt ist. Dies ist identisch zu Bounded Model-Checking mit Tiefe k. Für den Induktionsschritt wird überprüft, ob in einer beliebigen Schleifeniteration eine Spezifikationsverletzung existieren kann, falls in den vorhergehenden k Iterationen keine Spezifikationsverletzung aufgetreten ist (Induktionsannahme). Dies lässt sich umsetzen durch Bounded Model-Checking mit Tiefe $k+1$ und einem initialen Zustand am Kopf der Schleife des Programms anstatt am Programmanfang (das Verfahren lässt sich auf Programme mit mehreren Schleifen erweitern). Daher lässt sich auch dieses Verfahren mit Hilfe der Predicate CPA durchführen.

3.3 Prädikatenabstraktion

Späte Prädikatenabstraktion [GS97, He02] ist ein klassisches Verfahren für Model-Checking und wird durch die Predicate CPA unterstützt, indem für blk eine Wahl außer blk^{never} getroffen wird und die Prädikate-Verfeinerungsstrategie genutzt wird. Außerdem werden typischerweise die abstrakten Zustände durch BDDs repräsentiert, ansonsten ist die Konfiguration beliebig. Damit der $\mathsf{refine}_{\mathbb{P}}$-Operator zum Zuge kommt, muss ein Algorithmus für Gegenbeispiel-basierte Abstraktionsverfeinerung zusätzlich zum CPA-Algorithmus verwendet werden. Hierbei ist die Menge der Prädikate, die von $\mathsf{prec}_{\mathbb{P}}$ zur Abstraktionsberechnung verwendet werden, am Anfang leer, so dass jede Abstraktionsberechnung maximal überapproximiert. Bei unechten Gegenbeispielen werden dann durch $\mathsf{refine}_{\mathbb{P}}$ passende Prädikate gefunden und in die Menge der Prädikate eingefügt, wodurch zukünftige Abstraktionsberechnungen genauer werden und die unechten Gegenbeispiele ausgeschlossen werden können. Außerdem werden nach der Verfeinerung die zu ungenauen Teile des abstrakten Modells verworfen und neu berechnet.

3.4 IMPACT

Der IMPACT-Algorithmus [Mc06] ist ein Verfahren zur Gegenbeispiel-gesteuerten Abstraktionsverfeinerung, der gegenüber der Prädikatenabstraktion effizienter arbeiten soll, indem er die teuren Abstraktionsberechnungen vermeidet. Obwohl in der initialen Beschreibung komplett anders dargestellt, lässt er sich in unserem Rahmenwerk sehr ähnlich zur Prädikatenabstraktion ausdrücken: Die wichtigste Abweichung ist die IMPACT-Verfeinerungsstrategie, die das abstrakte Modell verfeinert, indem es die gefundenen abstrakten Fakten zu den vorhandenen abstrakten Zuständen hinzukonjugiert. Dadurch ist kein Verwerfen und Neuberechnen des abstrakten Modells nötig, und da die Menge der Prädikate für die Abstraktionsberechnung dauerhaft leer bleibt, bleibt die Abstraktionsberechnung dauerhaft eine triviale Operation. Allerdings werden nun statt BDDs SMT-Formeln zur Repräsentation der abstrakten Zustände verwendet, was dazu führt, dass Operationen wie der Vergleich von abstrakten Zuständen (für die Fixpunkterkennung nötig) teurer werden.

3.5 Anwendungen

Unser Rahmenwerk für prädikatenbasierte Software-Verifikation, das auf der Predicate CPA basiert, erlaubt es also so unterschiedliche Verfahren wie die genannten nur durch wenige Konfigurationsänderungen als Varianten auszudrücken. Die Unterschiede und Gemeinsamkeiten der Verfahren werden so hervorgehoben. Dies erlaubt es uns neue Erkenntnisse über die Kernideen dieser Verfahren zu erlangen [BW12]. Außerdem werden dadurch experimentelle Vergleichsstudien in einer neuen Qualität möglich [BDW17], da alle irrelevanten Unterschiede eliminiert wurden. Darüber hinaus können wir nun ohne weiteren Aufwand neue Verfahren definieren und damit experimentieren, indem wir die vorhandenen Verfahren und ihre Bestandteile neu kombinieren. Solche Ansätze werden in der Dissertation beschrieben [We17, Abschnitt 5.4].

4 Benchmarking für zuverlässige experimentelle Forschung

Die experimentelle Evaluation der Effizienz durch Benchmarking, d. h. das Messen von Zeit- und Speicherverbrauch eines Tools, ist eine effektive und günstige Methode um Forschungsergebnisse zu beurteilen. Im Bereich der Software-Verifikation ist dies die Standardmethode, z. B. um verschiedene Algorithmen oder Tools anhand ihrer Effizienz für eine große Menge an zu verifizierenden Programmen zu vergleichen. Im Rahmen der Dissertation wollen wir die in Abschnitt 3 vorgestellten Verfahren evaluieren und unsere Implementierung mit anderen Verifizierern vergleichen. Hierzu benötigen wir die Möglichkeit, den Zeit- und Speicherverbrauch einer Ausführung eines beliebigen Programms exakt zu messen. Um die Replizierbarkeit der Ergebnisse sicherzustellen, ist es darüber hinaus nötig, während der Programmausführung den Speicherverbrauch auf ein festgesetztes Maß zu limitieren und das Programm gegenüber äußeren Einflüssen abzuschirmen. Inbesondere Effekte, die zu einer nicht-deterministischen Zeitmessung führen können, müssen soweit wie möglich ausgeschlossen werden.

Drei Hauptprobleme müssen wir für zuverlässiges Benchmarking berücksichtigen. Erstens starten viele Programme Kindprozesse zur Erledigung bestimmter Aufgaben, deren Ressourcenverbrauch mit berücksichtigt werden muss. Kindprozesse zu nutzen ist eine gängige Praxis von Verifizierern, aber gängige Standardverfahren zum Messen des Zeitverbrauchs von Prozessen messen die Zeiten von Kindprozessen nicht zuverlässig mit. Dies kann zu Messfehlern beliebiger Größe führen. Zweitens kommt es zu Problemen, wenn aus Zeitgründen das Benchmarking mehrerer unabhängiger Programmausführungen parallel auf der gleichen Maschine nötig ist. Hardwareeigenschaften gängiger Systeme wie Hyper-Threading, Turbo Boost und Nonuniform Memory Access (NUMA) können dazu führen, dass der Zeitverbrauch eines Programms von den Aktivitäten parallel laufender Prozesse abhängt, selbst wenn jeder Prozess spezifische Hardwareeinheiten (z. B. CPU-Kerne) exklusiv zugewiesen bekommen hat. Daher muss beim Benchmarking darauf geachtet werden, die Zuteilung der Hardwareressourcen auf die Prozesse so vorzunehmen, dass diese Einflüsse ausgeschlossen oder zumindest minimiert werden. Die dritte Art von Problemen besteht darin, die Unabhängigkeit mehrerer (sequentieller oder paralleler) Programmausführungen zu gewährleisten, auch wenn das Programm z. B. temporäre Dateien

an bestimmte feste Orte schreibt. Insgesamt können wir die sechs in Abb. 2 aufgeführten Anforderungen identifizieren, die diese Probleme adressieren und die von einem Benchmarking-System sichergestellt werden müssen.

1. Messe und limitiere den Ressourcenverbrauch akkurat
2. Beende Prozesse zuverlässig
3. Weise CPU-Kerne gezielt zu
4. Respektiere Nonuniform Memory Access
5. Vermeide die Nutzung von Auslagerungsspeicher
6. Isoliere individuelle Programmausführungen

Abb. 2: Anforderungen für zuverlässiges Benchmarking

Diese Anforderungen können unter Linux nur umgesetzt werden durch die Verwendung von Kontrollgruppen („control groups“ oder „cgroups“) und Namensräumen („namespaces“), zwei Features des Linux-Kernels. Mit Kontrollgruppen lassen sich Prozesse in Gruppen zusammenfassen, und Ressourcen wie Zeit und Speicher lassen sich für solche Gruppen messen und limitieren. Mit Namensräumen lassen sich Prozesse in sogenannten „Containern“ isolieren, z. B. lässt sich verhindern, dass bestimmte Prozesse mit anderen Prozessen oder mit dem Netzwerk kommunizieren. Darüber hinaus lässt sich für Container die Sicht auf das Dateisystem modifizieren, so dass z. B. bestimmte Verzeichnisse unsichtbar sind oder jeder Container sein eigenes Verzeichnis für temporäre Dateien bekommt, um Wechselwirkungen zwischen Containern auszuschließen. Container sind ähnlich zu virtuellen Maschinen, allerdings ohne die Geschwindigkeitseinbußen und den höheren Speicherverbrauch von virtuellen Maschinen.

Da gängige Benchmarking-Programme die Anforderungen für zuverlässiges Benchmarking nicht einhalten, stellen wir das Tool BENCHEXEC [We17, BLW17] zur Verfügung. Es benutzt Kontrollgruppen und Namensräume, um die Einhaltung der Anforderungen aus Abb. 2 umzusetzen. BENCHEXEC ist unter der Open-Source-Lizenz Apache 2.0 auf GitHub verfügbar[4]. Es ist die technische Grundlage der International Competition on Software Verification (SV-COMP) [Be16], bei der über mehrere Jahre Millionen von Ausführungen mehrerer Dutzend Verifizierer durchgeführt und gemessen wurden. Dabei wäre es ohne die Techniken, auf denen BENCHEXEC basiert, stellenweise zu großen Messfehlern gekommen. Dies zeigt, dass zuverlässiges Benchmarking unersetzlich ist, um falsche wissenschaftliche Schlüsse zu vermeiden. Unser Tool BENCHEXEC ermöglicht dies.

5 Experimentelle Evaluation

Eine experimentelle Studie, die die vier in Abschnitt 3 vorgestellten Verfahren mit Hilfe unseres Rahmenwerks untereinander vergleicht, wurde bereits separat durchgeführt [BDW17], und lieferte interessante Erkenntnisse über die Effizienz und Effektivität der Verfahren für bestimmte Klassen von Eingabeprogrammen. In der Dissertation wird daher zum einen evaluiert, wie die erstellte Implementierung des Rahmenwerks in CPACHECKER im Vergleich

[4] `https://github.com/sosy-lab/benchexec`

mit anderen Verifizierern (basierend auf den gleichen und anderen Verfahren) abschneidet. Zum Vergleich werden die Ergebnisse der Int. Competition on Software Verification 2017 (SV-COMP'17) herangezogen, dem größten internationalen Wettbewerb für Software-Verifizierer mit 32 Teilnehmern im Jahr 2017. Der Vergleich wird für 5 594 C-Programme (mit und ohne bekannten Spezifikationsverletzungen) mit insgesamt 73 Millionen Zeilen Code (2.4 GB) ausgeführt. Wir zeigen, dass die Implementierung auf höchstem Niveau liegt, indem die verschiedenen Varianten der Predicate CPA jeweils vergleichbar viele Programme korrekt verifizieren wie die besten Teilnehmer an SV-COMP'17. In der Zwischenzeit wird dies bestätigt durch eine erfolgreiche Teilnahme an SV-COMP'18, in der CPACHECKER mit Hilfe dieser Implementierung den Sieg in der Kategorie Overall erzielte.

Zum anderen wurde mit den gleichen Eingabeprogrammen eine Studie mit 120 verschiedenen Konfigurationen der Predicate CPA durchgeführt – 30 Untervarianten für jedes der vier vorgestellten Verfahren. Diese Studie umfasste insgesamt 671 280 Ausführungen des Verifizierers und benötigte 3 620 Tage Rechenzeit. Die Studie zeigt, dass auch einige technische und kleinere konzeptionelle Entscheidungen, die in der Theorie unabhängig vom verwendeten Verifikationsverfahren sind, nicht nur signifikanten Einfluss auf die Effektivität und Effizienz haben, sondern auch Vergleiche zwischen den Verfahren beeinflussen können, da manche dieser Entscheidungen übermäßig nachteilig bei bestimmten Verfahren wirken. Die Ergebnisse der Studie liefern daher nicht nur wertvolle Informationen für Nutzer, die effektive und effiziente Verifikationsverfahren benötigen, sondern auch entscheidende Hinweise für Forscher, die diese bei experimentellen Studien beachten müssen. Ohne die Vereinheitlichung der Verifikationsverfahren in einem gemeinsamen flexiblen Rahmenwerk wäre diese Studie nicht möglich gewesen.

6 Zusammenfassung

Das in der Dissertation vorgestellte Rahmenwerk vereinheitlicht zum ersten Mal verschiedenste Verfahren zur automatischen Software-Verifikation und löst das Problem des fragmentierten Stands der Forschung, unterstützt durch eine robuste und effiziente Implementierung auf weltweit höchstem Niveau. Dies liefert fruchtbare Erkenntnisse und bildet die Grundlage für neue konzeptionelle und experimentelle Möglichkeiten sowohl in der Forschung als auch im praktischen Einsatz von prädikatenbasierten Analysen. Das vorgestellte neuartige Benchmarking-System ermöglicht mit Hilfe moderner Technologien zuverlässige experimentelle Forschung.

Sowohl das präsentierte Rahmenwerk als auch die entwickelten Tools haben sich bereits in einer Reihe von darauf aufbauenden Forschungsprojekten verschiedener Gruppen als hilfreich erwiesen. Die Verfahren und Tools der Dissertation werden regelmäßig zur Verifikation von Gerätetreibern des Linux-Kernels eingesetzt und haben dabei bereits geholfen Dutzende von Fehlern zu finden. Die in dem weltweit wichtigsten Wettbewerb für Software-Verifizierer erzielten Erfolge wurden mit der Kurt-Gödel-Medaille ausgezeichnet.

Literaturverzeichnis

[BDW17] Beyer, D.; Dangl, M.; Wendler, P.: A Unifying View on SMT-Based Software Verification. J. Autom. Reasoning, 2017.

[Be16] Beyer, D.: Reliable and Reproducible Competition Results with BENCHEXEC and Witnesses (SV-COMP 2016). In: Proc. TACAS. LNCS 9636. Springer, S. 887–904, 2016.

[BHT07] Beyer, D.; Henzinger, T. A.; Théoduloz, G.: Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In: Proc. CAV. LNCS 4590. Springer, S. 504–518, 2007.

[Bi99] Biere, A.; Cimatti, A.; Clarke, E. M.; Zhu, Y.: Symbolic Model Checking without BDDs. In: Proc. TACAS. LNCS 1579. Springer, S. 193–207, 1999.

[BK11] Beyer, D.; Keremoglu, M. E.: CPACHECKER: A Tool for Configurable Software Verification. In: Proc. CAV. LNCS 6806. Springer, S. 184–190, 2011.

[BLW17] Beyer, D.; Löwe, S.; Wendler, P.: Reliable Benchmarking: Requirements and Solutions. Int. J. Softw. Tools Technol. Transfer, 2017.

[BW12] Beyer, D.; Wendler, P.: Algorithms for Software Model Checking: Predicate Abstraction vs. IMPACT. In: Proc. FMCAD. FMCAD, S. 106–113, 2012.

[Cl03] Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM, 50(5):752–794, 2003.

[Cr57] Craig, W.: Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. J. Symb. Log., 22(3):250–268, 1957.

[DKR11] Donaldson, Alastair F.; Kroening, Daniel; Rümmer, Philipp: Automatic analysis of DMA races using model checking and *k*-induction. FMSD, 39(1):83–113, 2011.

[GS97] Graf, S.; Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Proc. CAV. LNCS 1254. Springer, S. 72–83, 1997.

[He02] Henzinger, T. A.; Jhala, R.; Majumdar, R.; Sutre, G.: Lazy abstraction. In: Proc. POPL. ACM, S. 58–70, 2002.

[Mc06] McMillan, K. L.: Lazy Abstraction with Interpolants. In: Proc. CAV. LNCS 4144. Springer, S. 123–136, 2006.

[We17] Wendler, Philipp: Towards Practical Predicate Analysis. Dissertation, Univ. Passau, 2017.

Philipp Wendler wurde am 29. April 1985 in Roth geboren. Er studierte von 2005 bis 2010 Informatik an der Universität Passau und erlangte in dieser Zeit sowohl den Bachelor- als auch den Master-Abschluss, jeweils mit Auszeichnung. Im Rahmen seiner Master-Arbeit, die mit dem IHK-Preis der IHK Niederbayern ausgezeichnet wurde, beschäftigte er sich erstmals mit Software-Verifikation. Anschließend vertiefte er die Forschung in diesem Bereich im Rahmen einer Promotion am Lehrstuhl von Prof. Dr. Dirk Beyer an der Universität Passau. Insbesondere beschäftigte er sich mit Software Model Checking, wo er ein Rahmenwerk für die Vereinheitlichung einer breiten Reihe von Ansätzen entwickelte. Er ist einer der Hauptentwickler des CPACHECKER-Projekts, eines der erfolgreichsten Software-Verifizierer weltweit. Für seine Arbeiten wurde er mit dem Young Scientist Award und der Kurt-Gödel-Medaille ausgezeichnet. Die Promotion schloss er 2017 mit Auszeichnung ab und seit 2016 arbeitet er bei Prof. Dr. Dirk Beyer an der LMU München.