Jahresbericht 2022 des Lehrstuhls für Informatik 2 (Programmiersysteme)

1 Mitarbeiterinnen und Mitarbeiter

Julian Brandner, M. Sc., Tobias Heineken, M. Sc. (seit 01.10.2022), Hon.-Prof. Dr.-Ing. Bernd Hindel, Hon.-Prof. Dr.-Ing. Detlef Kips, Patrick Kreutzer, M. Sc. (bis 30.11.2022), Florian Mayer, M. Sc., Dipl.-Inf. Daniela Novac (bis 31.03.2022), Dr.-Ing. Norbert Oster, Akad. ORat, Prof. Dr. Michael Philippsen (Ordinarius), Prof. em. Dr. Hans Jürgen Schneider (Emeritus), Dipl.-Ing. Frank Deserno (IT-Support), Margit Zenk (Sekretariat).

Gäste und externes Lehrpersonal am Lehrstuhl: Veronika Dashuber, M. Sc., Dr.-Ing. Klaudia Dussa-Zieger (Lehrbeauftragte), Tobias Feigl, M. Sc., Dr.-Ing. Martin Jung (Lehrbeauftragter).

2 Überblick

Wir liefern ingenieurwissenschaftliche Antworten für Software-Ingenieure, die **parallele Software** im industriellen Rahmen für **Multicore-Rechner**, für daraus bestehende verteilte Systeme, sowie für vernetzte eingebettete Systeme entwickeln. Wir arbeiten **Programm-Code-basiert**, erstellen lauffähige **Prototypen** und **evaluieren** diese quantitativ und qualitativ. Eckpunkte unserer Forschungsthemen:
(a) Wir arbeiten an **Programmiermodellen** für **heterogene** Parallelität und erzeugen dafür portablen und effizienten Code für Multicores, GPUs, Acceleratoren, Mobile Geräte, FPGAs u.ä.
(b) Wir unterstützen die **Parallelisierung** von Software für Multicore-Rechner. Unsere Werkzeuge analysieren **Code-Repositories** und helfen dem Entwickler bei der **Migration** und **Refaktorisierung**.
(c) Wir analysieren Programme. Unsere **Code-Analysewerkzeuge** sind schnell, interaktiv, inkrementell und arbeiten teilweise selbst parallel. Sie finden Wettlaufsituationen, konkurrierende Ressourcenzugriffe etc. im Code und zeigen dem Entwickler Verbesserungsvorschläge punktgenau und in der Entwicklungsumgebung an.
(d) Wir **testen** parallelen Code und **diagnostizieren** Problemursachen. Unsere Werkzeuge erzeugen Testdaten, finden Ursachen von erratischem Laufzeitverhalten und schützen gegen **Authentizitätsangriffe**.

3 Forschung

3.1 AnaCoRe – *Analyse von Code-Repositories*

Bei der Weiterentwicklung von Software führen die Entwickler oftmals sich wiederholende, ähnliche Änderungen durch. Dazu gehört beispielsweise die Anpassung von Programmen an eine veränderte Bibliotheksschnittstelle, die Behebung von Fehlern in funktional ähnlichen Komponenten sowie die Parallelisierung von sequentiellen Programmteilen. Wenn jeder Entwickler die nötigen Änderungen selbst erarbeiten muss, führt dies leicht zu fehlerhaften Programmen, beispielsweise weil weitere zu ändernde Stellen übersehen werden. Wünschenswert wäre stattdessen ein automatisiertes Verfahren, das ähnliche Änderungen erkennt und mit dieser Wissensbasis Software-Entwickler bei weiteren Änderungen unterstützt.

Änderungsextraktion
In 2017 entwickelten wir ein neues Vorschlagssystem mit Namen ARES (Accurate REcommendation System). Verglichen mit bisherigen Ansätzen erzeugt es genauere Vorschläge, da seine Algorithmen Code-Verschiebungen während der Muster- und Vorschlagserzeugung berücksichtigen. Der Ansatz basiert darauf, dass zwei Versionen eines Programms miteinander verglichen werden. Das Werkzeug extrahiert dabei automatisch, welche Änderungen sich zwischen den beiden Versionen ergeben haben, und leitet daraus generalisierte Muster aus zu ersetzenden Code-Sequenzen ab. Diese Muster können anschließend von ARES dazu verwendet werden, analoge Änderungen für den Quellcode anderer Programme automatisch vorzuschlagen.
Zur Extraktion der Änderungen verwenden wir ein baumbasiertes Verfahren. Im Jahr 2016 wurde ein neuer Algorithmus (MTDIFF) für solche baumbasierten Verfahren entwickelt und gut sichtbar publiziert, der die Genauigkeit der Änderungsbestimmung verbessert.

Symbolische Ausführung von Code-Fragmenten
Im Jahr 2014 wurde ein neues Verfahren zur symbolischen Code-Ausführung namens SYFEX entwickelt, welches die Ähnlichkeit des Verhaltens zweier Code-Teilstücke bestimmt. Mit diesem Verfahren soll eine Steigerung der Qualität der Verbesserungsvorschläge erreicht werden. Abhängig von der Anzahl und Generalität der Muster in der Datenbank kann SIFE ohne das neue Verfahren unpassende Vorschläge liefern. Um dem Entwickler nur die passenden Vorschläge anzuzeigen, wird das semantische Verhalten des Vorschlags mit dem semantischen Verhalten des Musters aus der Datenbank verglichen. Weichen beide zu sehr voneinander ab, wird der Vorschlag aus der Ergebnismenge entfernt. Die Besonderheit von SYFEX besteht darin, dass es auf herausgelöste Code-Teilstücke anwendbar ist und keine menschliche Vorkonfiguration benötigt.
SYFEX wurde im Jahr 2015 verfeinert und auf Code-Teilstücke aus Archiven von verschiedenen Software-Projekten angewendet. Der Schwerpunkt im Jahr 2016 lag auf einer Untersuchung, inwieweit SYFEX zum semantischen Vergleich von Abgaben eines Programmierwettbewerbs geeignet ist. In den Jahren 2017 und 2018 wurde SYFEX optimiert. Des Weiteren wurde mit der Erstellung eines Datensatzes semantisch ähnlicher Methoden aus quelloffenen Software-Archiven begonnen, der im Jahr 2019 veröffentlicht wurde.
Verfahren zur symbolischen Ausführung beruhen auf Algorithmen zur Erfüllbarkeitsprüfung von logisch-mathematischen Ausdrücken, um zulässige Ausführungspfade in einem Programm zu bestimmen. Oftmals beanspruchen diese Algorithmen einen großen Teil der aufgewendeten Rechenzeit. Um diese Erfüllbarkeitsprüfung zu beschleunigen, wurde in den Jahren 2019 und 2020 mit einer Technik experimentiert, um komplizierte Ausdrücke durch einfachere Ausdrücke mit gleicher Bedeutung zu ersetzen. Hierbei werden die einfacheren Ausdrücke durch ein Verfahren zur Programmsynthese aufgedeckt. Im Jahr 2020 wurde diese Programmsynthese um ein neuartiges Verfahren ergänzt, das für eine bestimmte Menge an Operationen bereits vorab ermitteln kann, ob sich damit ein Ausdruck mit gleicher Bedeutung wie der kompliziertere Quellausdruck bilden lässt. Unsere im Jahr 2021 erschienene wissenschaftliche Publikation beschreibt dieses Verfahren und zeigt, dass durch dessen Einsatz die Laufzeit von gängigen Programmsynthetisierern im Mittel um 33% verringert werden kann. Ebenfalls im Jahr 2021 wurde das Verfahren auf weitere Klassen von Programmsyntheseproblemen erweitert. Im Jahr 2022 wurden diese Erweiterungen umfangreich evaluiert. Diese Evaluation zeigte, dass die Erweiterungen zu einer vergleichbaren Beschleunigung gängiger Programmsyntheseverfahren auf einer größeren Klasse von Syntheseproblemen führen.

Detektion von semantisch ähnlichen Code-Fragmenten
SYFEX erlaubt es, die semantische Ähnlichkeit zweier Code-Fragmente zu bestimmen. So ist es damit prinzipiell möglich, Paare oder Gruppen von semantisch ähnlichen Code-Fragmenten (semantische Klone) zu identifizieren. Auf Grund des hohen Laufzeitaufwands verbietet sich der Einsatz von SYFEX – wie auch von anderen Werkzeugen dieser Art – allerdings, um in größeren Code-Projekten nach seman-

tisch ähnlichen Code-Fragmenten zu suchen. Im Jahr 2016 wurde deshalb mit der Entwicklung eines Verfahrens begonnen, mit dessen Hilfe die Detektion semantisch ähnlicher Code-Fragmente beschleunigt werden kann. Grundlage dieses Verfahrens ist eine Reihe von sog. Basiskomparatoren, die zwei Code-Fragmente jeweils hinsichtlich eines Kriteriums (beispielsweise die Anzahl bestimmter Kontrollstrukturen oder die Beschaffenheit der Kontrollflussgraphen) miteinander vergleichen und dabei möglichst geringen Laufzeitaufwand haben. Diese Basiskomparatoren können anschließend zu einer Hierarchie von Verfahren verknüpft werden. Um damit die semantische Ähnlichkeit zweier Fragmente möglichst genau bestimmen zu können, wird mit Hilfe der Genetischen Programmierung nach Hierarchien gesucht, die die von SYFEX für eine Reihe von Code-Paaren berechneten Ähnlichkeitswerte möglichst gut approximieren. Im Rahmen einer ersten Untersuchung hat sich gezeigt, dass sich das implementierte Verfahren tatsächlich für die Bestimmung von semantisch ähnlichen Code-Paaren eignet.
Die Implementierung dieses Verfahrens wurde in den Jahren 2017 und 2018 weiter verbessert. Zudem spielte die tiefergehende Evaluation des Verfahrens auf Basis von Methodenpaaren aus Software-Archiven sowie von Abgaben für Programmieraufgaben eine wichtige Rolle.

Semantische Code-Suche
Häufig steht die bei der Software-Entwicklung zu implementierende Funktionalität bereits in ähnlicher Form als Teil von Programmbibliotheken zur Verfügung. In vielen Fällen ist es ratsam, diese bereits vorhandene Realisierung zu verwenden statt die Funktionalität erneut zu implementieren, beispielsweise um den Aufwand für das Entwickeln und Testen des Codes zu reduzieren.
Voraussetzung für die Wiederverwendung einer für den Anwendugszweck geeigneten Implementierung ist, dass Entwickler diese überhaupt finden können. Zu diesem Zweck werden bereits heute regelmäßig Code-Suchmaschinen verwendet. Etablierte Verfahren stützen sich dabei insbesondere auf syntaktische Merkmale, d.h. der Nutzer gibt beispielsweise eine Reihe von Schlüsselwörtern oder Variablen- und Methodennamen an, nach denen die Suchmaschine suchen soll. Bei diesen Verfahren bleibt die Semantik des zu suchenden Codes unberücksichtigt. Dies führt in der Regel dazu, dass relevante, aber syntaktisch verschiedene Implementierungen nicht gefunden werden („false negatives“) oder dass syntaktisch ähnliche, aber semantisch irrelevante Ergebnisse präsentiert werden („false positives“). Die Suche nach Code-Fragmenten auf Basis ihrer Semantik ist Gegenstand aktueller Forschung.
Im Jahr 2017 wurde am Lehrstuhl mit der Entwicklung eines neuen Verfahrens zur semantischen Code-Suche begonnen. Der Nutzer spezifiziert dabei die gesuchte Funktionalität in Form von Eingabe-Ausgabe-Beispielen. Mit Hilfe eines aus der Literatur stammenden Verfahrens zur Funktionssynthese wird eine Methode erzeugt, die das durch die Beispiele beschriebene Verhalten möglichst genau realisiert. Diese synthetisierte Methode wird dann mit Hilfe des im Rahmen dieses Forschungsprojekts entwickelten Verfahrens zur Detektion von semantisch ähnlichen Code-Fragmenten mit den Methodenimplementierungen vorgegebener Programmbibliotheken verglichen, um ähnliche Implementierungen zu finden, die dem Nutzer als Ergebnis der Suche präsentiert werden. Eine erste Evaluation der prototypischen Implementierung zeigt die Umsetzbarkeit und Verwendbarkeit des Verfahrens.

Cluster-Bildung von ähnlichen Code-Änderungen
Voraussetzung für die Erzeugung generalisierter Änderungsmuster ist es, die Menge aller aus einem Quelltext-Archiv extrahierten Code-Änderungen in Teilmengen zueinander ähnlicher Änderungen aufzuteilen. Im Jahr 2015 wurde diese Erkennung ähnlicher Änderungen im Rahmen eines neuen Werkzeugs C3 verbessert. In einem ersten Schritt wurden verschiedene Metriken für den paarweisen Ähnlichkeitsvergleich der extrahierten Code-Änderungen implementiert und evaluiert. Darauf aufbauend wurden aus der Literatur bekannte Clustering-Algorithmen evaluiert und neue Heuristiken zur automatisierten Bestimmung der jeweiligen Parameter implementiert, um das bisherige naive Verfahren zur Identifizierung ähnlicher Änderungen zu ersetzen. Mit den im Rahmen von C3 implementierten Verfahren konnte im Vergleich zum bisherigen Ansatz eine deutliche Verbesserung erzielt werden. So können mit den neuen Verfahren mehr Gruppen ähnlicher Änderungen identifiziert werden, die sich für die Weiterverarbeitung

im Rahmen von SIFE zur Generierung von Vorschlägen eignen.
Die zweite Verbesserung zielt darauf ab, die erhaltenen Gruppen ähnlicher Änderungen zusätzlich automatisiert zu verfeinern. Zu diesem Zweck wurden verschiedene Verfahren aus dem Umfeld des maschinellen Lernens zur Ausreißererkennung untersucht, um Änderungen, die fälschlicherweise einer Gruppe zugeordnet wurden, wieder zu entfernen.
Im Jahr 2016 wurde C3 um eine weitere Metrik zum Vergleich zweier Code-Änderungen erweitert, die im Wesentlichen den textuellen Unterschied zwischen den Änderungen (wie er beispielsweise von dem Unix-Werkzeug 'diff' erzeugt wird) bewertet. Des Weiteren wurde das in C3 implementierte Verfahren im Rahmen eines Konferenzbeitrags veröffentlicht. In diesem Zusammenhang wurde auch der zur Evaluation des Verfahrens erzeugte Datensatz von Gruppen ähnlicher Änderungen unter einer Open-Source-Lizenz veröffentlicht, siehe https://github.com/FAU-Inf2/cthree. Dieser kann zukünftigen Arbeiten als Referenz oder Eingabe dienen. Außerdem wurden prototypisch Verfahren implementiert, mit denen die Ähnlichkeitsberechnung und das Clustering in C3 inkrementell erfolgen können. Diese erlauben es, dass bei neuen Änderungen, die zu einem Software-Archiv hinzugefügt werden, die zuvor bereits berechneten Ergebnisse weiterverwendet werden können und nur ein Teil der Arbeit wiederholt werden muss.

3.2 AutoCompTest – *Automatisiertes Testen von Übersetzern*

Übersetzer für Programmiersprachen sind äußerst komplexe Anwendungen, an die hohe Korrektheitsanforderungen gestellt werden: Ist ein Übersetzer fehlerhaft (d.h. weicht sein Verhalten vom dem durch die Sprachspezifikation definierten Verhalten ab), so generiert dieser u.U. fehlerhaften Code oder stürzt bei der Übersetzung mit einer Fehlermeldung ab. Solche Fehler in Übersetzern sind oftmals schwer zu bemerken oder zu umgehen. Nutzer erwarten deshalb i.A. eine (möglichst) fehlerfreie Implementierung des verwendeten Übersetzers.

Leider lassen sowohl vergangene Forschungsarbeiten als auch Fehlerdatenbanken im Internet vermuten, dass kein real verwendeter Übersetzer fehlerfrei ist. Es wird deshalb an Ansätzen geforscht, mit deren Hilfe die Qualität von Übersetzern gesteigert werden kann. Da die formale Verifikation (also der Beweis der Korrektheit) in der Praxis oftmals nicht möglich oder rentabel ist, zielen viele der Forschungsarbeiten darauf ab, Übersetzer möglichst umfangreich und automatisiert zu testen. In den meisten Fällen erhält der zu testende Übersetzer dabei ein Testprogramm als Eingabe. Anschließend wird das Verhalten des Übersetzers bzw. des von ihm generierten Programms überprüft: Weicht dieses vom erwarteten Verhalten ab (stürzt der Übersetzer also beispielsweise bei einem gültigen Eingabeprogramm mit einer Fehlermeldung ab), so wurde ein Fehler im Übersetzer gefunden. Soll dieser Testvorgang automatisiert stattfinden, ergeben sich zwei wesentliche Herausforderungen:

- Woher kommen die Testprogramme, auf die der Übersetzer angewendet wird?
- Was ist das erwartete Verhalten des Übersetzers bzw. des von ihm erzeugten Codes? Wie kann bestimmt werden, ob das tatsächliche Verhalten des Übersetzers korrekt ist?

Während die wissenschaftliche Literatur diverse Lösungen für die zweite Herausforderung vorstellt, die auch in der Praxis bereits etabliert sind, stellt die automatisierte Generierung zufälliger Testprogramme noch immer eine große Hürde dar. Damit Testprogramme zur Detektion von Fehlern in allen Teilen des Übersetzers verwendet werden können, müssen diese allen Regeln der jeweiligen Programmiersprache genügen, d.h. die Programme müssen syntaktisch und semantisch korrekt (und damit übersetzbar) sein. Auf Grund der Vielzahl an Regeln „echter“ Programmiersprachen stellt die Generierung solcher übersetzbarer Programme eine schwierige Aufgabe dar. Dies wird zusätzlich dadurch erschwert, dass das Programmgenerierungsverfahren möglichst effizient arbeiten muss: Die wissenschaftliche Literatur zeigt, dass die Effizienz eines solchen Verfahrens maßgebliche Auswirkungen auf seine Effektivität hat – nur

wenn in kurzer Zeit viele (und große) Programme generiert werden können, kann das Verfahren sinnvoll zur Detektion von Übersetzerfehlern eingesetzt werden.

In der Praxis scheitert das automatisierte Testen von Übersetzern deshalb oftmals daran, dass kein zugeschnittener Programmgenerator verfügbar ist und die Entwicklung eines solchen einen zu hohen Aufwand bedeutet. Ziel unseres Forschungsprojekts ist daher die Entwicklung von Verfahren, die den Aufwand für die Implementierung von effizienten Programmgeneratoren reduzieren.

Im Jahr 2018 haben wir mit der Entwicklung eines entsprechenden Werkzeugs begonnen. Als Eingabe dient eine Spezifikation der syntaktischen und semantischen Regeln der jeweiligen Programmiersprache in Form einer abstrakten Attributgrammatik. Eine solche erlaubt eine knappe Notation der Regeln auf hohem Abstraktionsniveau. Ein von uns neu entwickelter Algorithmus erzeugt dann Testprogramme, die allen spezifizierten Regeln genügen. Der Algorithmus nutzt dabei diverse technische Ideen aus, um eine angemessene Laufzeit zu erreichen. Dies ermöglicht die Generierung großer Testfallmengen in vertretbarer Zeit, auch auf üblichen Arbeitsplatzrechnern. Eine erste Evaluation hat nicht nur gezeigt, dass unser Verfahren sowohl effektiv als auch effizient ist, sondern auch dass es flexibel einsetzbar ist. So haben wir mit Hilfe unseres Verfahrens nicht nur Fehler in den C-Übersetzern gcc und clang entdeckt (unser Verfahren erreicht dabei eine ähnliche Fehleraufdeckungsgüte wie ein sprachspezifischer Programmgenerator aus der wissenschaftlichen Literatur), sondern auch diverse Bugs in mehreren SMT-Entscheidern. Einige der von uns entdeckten Fehler waren den jeweiligen Entwicklern zuvor noch unbekannt.

Im Jahr 2019 haben wir zusätzliche Features für das Schreiben von Sprachspezifikationen implementiert und die Effizienz des Programmgenerierungsverfahrens gesteigert. Durch die beiden Beiträge konnte der Durchsatz unseres Werkzeugs deutlich gesteigert werden. Des Weiteren haben wir mit Hilfe neuer Sprachspezifikationen Fehler in Übersetzern für die Programmiersprachen Lua und SQL aufgedeckt. Die Ergebnisse unserer Arbeit sind in eine Ende 2019 eingereichte (und inzwischen angenommene) wissenschaftliche Publikation eingeflossen. Neben der Arbeit an unserem Verfahren zur Programmgenerierung haben wir außerdem mit der Arbeit an einem Verfahren zur Testfallreduzierung begonnen. Das Verfahren reduziert die Größe eines zufällig generierten Testprogramms, das einen Fehler in einem Übersetzer auslöst, um die Suche nach der Ursache des Fehlers zu vereinfachen.

Im Jahr 2020 lag der Fokus des Forschungsprojekts auf sprachunabhängigen Verfahren zur automatischen Testfallreduzierung. Die wissenschaftliche Literatur schlägt unterschiedliche Reduzierungsverfahren vor. Da es bislang allerdings keinen aussagekräftigen Vergleich dieser Verfahren gibt, ist unklar, wie effizient und effektiv die vorgeschlagenen Reduzierungsverfahren tatsächlich sind. Wie wir festgestellt haben, gibt es dafür im Wesentlichen zwei Gründe, die darüber hinaus auch die Entwicklung und Evaluation neuer Verfahren erschweren. Zum einen verwenden die vorhandenen Implementierungen der vorgeschlagenen Reduzierungsverfahren unterschiedliche Implementierungssprachen, Programmrepräsentationen und Eingabegrammatiken. Mit diesen Implementierungen ist ein fairer Vergleich dieser Verfahren deshalb kaum möglich. Zum anderen gibt es keine umfangreiche Sammlung von (noch unreduzierten) Testprogrammen zur Evaluation von Reduzierungsverfahren. Dies hat zur Folge, dass die publizierten Reduzierungsverfahren jeweils nur mit sehr wenigen Testprogrammen evaluiert wurden, was die Aussagekraft der präsentierten Ergebnisse beeinträchtigt. Da in manchen Fällen darüber hinaus nur Testprogramme in einer einzigen Programmiersprache verwendet wurden, ist bislang unklar, wie gut diese Verfahren für Testprogramme in anderen Programmiersprachen funktionieren (wie sprachunabhängig diese Verfahren also tatsächlich sind). Um diese Lücken zu schließen, haben wir im Jahr 2020 mit der Implementierung eines Frameworks begonnen, das die wichtigsten Reduzierungsverfahren beinhaltet und so einen fairen Vergleich dieser Verfahren ermöglicht. Außerdem haben wir mit der Erstellung einer Testfallsammlung begonnen. Diese beinhaltet bereits etwa 300 Testprogramme in den Sprachen C und SMT-LIB 2, die etwa 100 unterschiedliche Fehler in realen Übersetzern auslösen. Diese Testfallsammlung erlaubt nicht nur aussagekräftigere Vergleiche von Reduzierungsverfahren, sondern verringert außerdem den Aufwand für

die Evaluation zukünftiger Verfahren. Anhand erster Experimente konnten wir feststellen, dass es bislang kein Reduzierungsverfahren gibt, dass in allen Fällen am besten geeignet ist.

Außerdem haben wir uns im Jahr 2020 mit der Frage beschäftigt, wie das im Rahmen des Forschungsprojekts entstandene Framework zur Generierung zufälliger Testprogramme erweitert werden kann, um neben funktionalen Fehlern auch Performance-Probleme in Übersetzern finden zu können. Im Rahmen einer Abschlussarbeit ist dabei ein Verfahren entstanden, das eine Menge zufällig generierter Programme mit Hilfe von Optimierungstechniken schrittweise so verändert, dass die resultierenden Programme im getesteten Übersetzer deutlich höhere Laufzeiten als in einer Referenzimplementierung auslösen. Erste Experimente haben gezeigt, dass so tatsächlich Performance-Probleme in Übersetzern gefunden werden können.

Im Jahr 2021 haben wir die Implementierung der wichtigsten Reduzierungsverfahren aus der wissenschaftlichen Literatur sowie die Erstellung einer Testfallsammlung für deren Evaluation abgeschlossen. Aufbauend darauf haben wir außerdem einen quantitativen Vergleich der Verfahren durchgeführt; soweit wir wissen, handelt es sich dabei um den mit Abstand umfangreichsten und aussagekräftigsten Vergleich bisheriger Reduzierungsverfahren. Unsere Ergebnisse zeigen, dass es bislang kein Verfahren gibt, das in allen Anwendungsfällen am besten geeignet wäre. Außerdem konnten wir feststellen, dass es für alle Verfahren zu deutlichen Ausreißern kommen kann, und zwar sowohl hinsichtlich Effizienz (also wie schnell ein Reduzierungsverfahren ein Eingabeprogramm reduzieren kann) als auch Effektivität (also wie klein das Ergebnis eines Reduzierungsverfahrens ist). Dies deutet darauf hin, dass es noch Potenzial für die Entwicklung weiterer Reduzierungsverfahren in der Zukunft gibt, und unsere Ergebnisse liefern einige Einsichten, was dabei zu beachten ist. So hat sich beispielsweise gezeigt, dass ein Hochziehen von Knoten im Syntaxbaum unabdingbar für die Generierung möglichst kleiner Ergebnisse (und damit eine hohe Effektivität) ist und dass eine effiziente Behandlung von Listenstrukturen im Syntaxbaum notwendig ist. Die Ergebnisse unserer Arbeit sind in eine im Jahr 2021 eingereichte und angenommene Publikation eingeflossen.

Außerdem haben wir im Jahr 2021 untersucht, ob bzw. wie sich die Effektivität unseres Programmgenerierungsverfahrens steigern lässt, wenn bei der Generierung die Überdeckung der zugrundeliegenden Grammatik berücksichtigt wird. Im Rahmen einer Abschlussarbeit wurden dazu unterschiedliche, aus der wissenschaftlichen Literatur stammende kontextfreie Überdeckungsmetriken für den Anwendungsfall adaptiert sowie implementiert und evaluiert. Dabei hat sich gezeigt, dass die Überdeckung hinsichtlich einer kontextfreien Metrik nur bedingt mit der Fehleraufdeckung korreliert. In zukünftigen Arbeiten sollte deshalb untersucht werden, ob Überdeckungsmetriken, die auch kontextsensitive, semantische Eigenschaften berücksichtigen, besser für diesen Anwendungsfall geeignet sind.

Im Jahr 2022 wurde im Rahmen einer Abschlussarbeit mit der Entwicklung eines Rahmenwerks für die Realisierung sprachadaptierter Reduktionsverfahren begonnen. Dieses Rahmenwerk stellt eine domänenspezifische Sprache (DSL) zur Verfügung, mit deren Hilfe sich Reduktionsverfahren auf einfache und knappe Art und Weise beschreiben lassen. Mit diesem Rahmenwerk und der entwickelten DSL soll es möglich sein, bestehende Reduktionsverfahren mit möglichst wenig Aufwand an die Besonderheiten einer bestimmten Programmiersprache anpassen zu können. Die Hoffnung dabei ist, dass solche sprachadaptierten Verfahren noch effizienter und effektiver arbeiten können als die bestehenden, sprachunabhängigen Reduktionsverfahren. Darüber hinaus soll das Rahmenwerk auch den Aufwand für die Entwicklung zukünftiger Reduktionsverfahren verringern und könnte so einen wertvollen Beitrag für die Forschung auf diesem Gebiet leisten.

3.3 Holoware – *Kooperative Exploration und Analyse von Software in einer Virtual/Augmented Reality Appliance*

Der Aufwand für das Verstehen von Software umfasst in Entwicklungsprojekten bis zu 30% und in Wartungsprojekten bis zu 80% der Programmieraufwände. Deshalb wird in modernen Arbeitsumgebungen zur Software-Entwicklung eine effiziente und effektive Möglichkeit zum Software-Verstehen benötigt. Die dreidimensionale Visualisierung von Software steigert das Verständnis der Sachverhalte deutlich, und damit liegt eine Nutzung von Virtual-Reality-Techniken nahe. Im Rahmen des Holoware Projekts schaffen wir eine Umgebung, in der Software mit Hilfe von VR/AR (Virtual/Augmented Reality) und Technologien der Künstlichen Intelligenz (KI) kooperativ exploriert und analysiert werden kann. In dieser virtuellen Realität wird ein Software-Projekt oder -verbund dreidimensional visualisiert, sodass mehrere Benutzer gleichzeitig die Software gemeinsam und kooperativ erkunden und analysieren können. Verschiedene Nutzer können dabei aus unterschiedlichen Perspektiven und mit unterschiedlich angereicherten Sichten profitieren und erhalten so einen intuitiven Zugang zur Struktur und zum Verhalten der Software. Damit sollen verschiedene Nutzungsszenarien möglich sein, wie z.B. die Anomalieanalyse im Expertenteam, bei der mehrere Domänenexperten gemeinsam eine Laufzeitanomalie der Software analysieren. Sie sehen dabei die selbe statische Struktur der Software, jeder Experte jedoch angereichert mit den für ihn relevanten Detail-Informationen. Im VR-Raum können sie ihre Erkenntnisse kommunizieren und so ihre unterschiedliche Expertise einbringen.

Darüber hinaus werden die statischen und dynamischen Eigenschaften des Software-Systems analysiert. Zu den statischen Eigenschaften zählen beispielsweise der Source-Code, statische Aufrufbeziehungen oder auch Metriken wie LoC, zyklomatische Komplexität o. Ä. Dynamische Eigenschaften lassen sich in Logs, Ablaufspuren (Traces), Laufzeitmetriken oder auch Konfigurationen, die zur Laufzeit eingelesen werden, gruppieren. Die Herausforderung liegt darin, diese Vielzahl an Informationen zu aggregieren, analysieren und korrelieren. Es wird eine Anomalie- und Signifikanz-Detektion entwickelt, die sowohl Struktur- als auch Laufzeitauffälligkeiten automatisch erkennt. Zudem wird ein Vorhersagesystem aufgebaut, das Aussagen über die Komponentengesundheit macht. Dadurch kann beispielsweise vorhergesagt werden, welche Komponente gefährdet ist, demnächst auszufallen. Bisher werden die Ablaufspuren um die Log-Einträge angereichert, wodurch ein detailliertes Bild der dynamischen Aufrufbeziehungen entsteht. Diese dynamischen Beziehungen werden auf den statischen Aufrufgraph abgebildet, da sie Aufrufe beschreiben, die aus der statischen Analyse nicht hervorgehen (beispielsweise REST-Aufrufe über mehrere verteilte Komponenten).

Im Jahr 2018 konnten folgende wesentlichen Beiträge geleistet werden:

- Entwicklung eines funktionsfähigen VR-Visualisierungsprototyps zu Demonstrations- und Forschungszwecken.
- Mapping von dynamischer Laufzeitdaten auf die statische Struktur als Grundlage für deren Analyse und Visualisierung.
- Entwurf und Implementierung der Anomalieerkennung von Ablaufspuren durch ein Unsupervised-Learning-Verfahren.

Im Jahr 2019 konnten weitere Verbesserungen erreicht werden:

- Erweiterung des Prototyps um die Darstellung dynamischen Software-Verhaltens.
- Kooperative (Remote-)Nutzung des Visualisierungsprototyps.
- Auswertung von Commit-Nachrichten zur Anomalieerkennung.
- Clustering der Aufrufe eines Systems nach Anwendungsfällen.

In dem Papier „Towards Collaborative and Dynamic Software Visualization in VR“, das auf der International Conference on Computer Graphics Theory and Applications (VISIGRAPP) 2020 angenommen wurde, haben wir die Wirksamkeit unseres Prototyps zur Effizienzsteigerung beim Software-Verstehen gezeigt. Im Jahr 2020 wurde unser Papier „A Layered Software City for Dependency Visualization“ auf der International Conference on Computer Graphics Theory and Applications (VISIGRAPP) 2021 angenommen und mit dem „Best Paper“-Award ausgezeichnet. Wir konnten belegen, dass das von uns entwickelte Layered Layout für Software-Städte das Analysieren von Software-Architektur vereinfacht und das Standard-Layout bei weitem übertrifft. Der finale Prototyp und die Publikationen, die im Rahmen des Forschungsprojektes entstanden sind, führten zu einem erfolgreichen Projektabschluss.

Nach Auslaufen der offiziellen Projektförderung durften wir in 2021 eine erweiterte Version des Award-Papiers („Static And Dynamic Dependency Visualization in a Layered Software City“) als Zeitschriftenartikel zur Begutachtung einreichen. Hier stellen wir eine Nacht-Ansicht der Stadt vor, in der die dynamischen Aufrufbeziehungen als Bögen visualisiert werden. Wir widmeten uns also einem zentralen, noch offenen Punkt: der Visualisierung von dynamischen Abhängigkeiten. In dem Papier „Trace Visualization within the Software City Metaphor: A Controlled Experiment on Program Comprehension“ auf der IEEE Working Conference on Software Visualization (VISSOFT) haben wir dynamische Abhängigkeiten innerhalb der Software-Stadt über Licht-Intensitäten aggregiert dargestellt und konnten zeigen, dass diese Darstellung hilfreicher ist als alle Abhängigkeiten zu zeichnen. Auch für dieses Papier wurden wir zur Einreichung eines erweiterten Artikels „Trace Visualization within the Software City Metaphor: Controlled Experiments on Program Comprehension“ zur Begutachtung aufgefordert. Wir zeigen dort eine erweiterte Darstellung dynamischer Abhängigkeiten und färben Bögen basierend auf HTTP Statuscodes.

In 2022 wurden beide Journalbeiträge akzeptiert: „Static And Dynamic Dependency Visualization in a Layered Software City“ ist im Springer Nature Computer Science Journal veröffentlicht und „Trace Visualization within the Software City Metaphor: Controlled Experiments on Program Comprehension“ wurde für das Information and Software Technology Journal angenommen. Zur Finalisierung von Holoware wurden alle Erweiterungen zu einer Gesamtvisualisierung zusammengefasst. Dazu wurden unterschiedlichen Ansichten verwendet, zwischen denen der Nutzer umschalten kann: in der Tagesansicht kann die Software-Architektur im neuartigen Holoware-Schichten-Layout analysiert werden und in der Nachtansicht werden dynamische Abhängigkeiten dargestellt. Im Rahmen einer Abschlussarbeit wurde Holoware zudem als AR-Visualisierung umgesetzt, sodass sie leicht als Showcase oder im Arbeitsalltag eingesetzt werden kann.

3.4 ORKA-HPC – *OpenMP für rekonfigurierbare heterogene Architekturen*

High-Performance Computing (HPC) ist ein wichtiger Bestandteil für die europäische Innovationskapazität und wird auch als ein Baustein bei der Digitalisierung der europäischen Industrie gesehen. Rekonfigurierbare Technologien wie Field Programmable Gate Array (FPGA) Module gewinnen hier wegen ihrer Energieeffizienz, Performance und ihrer Flexibilität immer größere Bedeutung.
Es wird außerdem zunehmend auf HPC-Systeme mit heterogenen Architekturen gesetzt, auch auf solche mit FPGA-Beschleunigern. Die große Flexibilität dieser FPGAs ermöglicht es, dass eine große Klasse von HPC-Applikationen mit FPGAs realisiert werden kann. Allerdings ist deren Programmierung bisher vorwiegend Spezialisten vorbehalten und sehr zeitaufwendig, wodurch deren Verwendung in Bereichen des wissenschaftlichen Höchstleistungsrechnens derzeit noch selten ist.
Im HPC-Umfeld gibt es verschiedenste Programmiermodelle für heterogene Rechnersysteme mit einigen Typen von Beschleunigern. Gängige Programmiermodelle sind zum Beispiel OpenCL (opencl.org), OpenACC (openacc.org) und OpenMP (OpenMP.org). Eine produktive Verwendbarkeit dieser Standards für FPGAs ist heute jedoch noch nicht gegeben.

Ziele des ORKA Projektes sind:

1. Nutzung des OpenMP-4.0-Standards als Programmiermodell, um ohne Spezialkenntnisse heterogene Rechnerplattformen mit FPGAs als rekonfigurierbare Architekturen durch portable Implementierungen eine breitere Community im HPC-Umfeld zu erschließen.
2. Entwurf und Implementierung eines Source-to-Source-Frameworks, welches C/C++-Code mit OpenMP-4.0-Direktiven in ein ausführbares Programm transformiert, das die Host-CPUs und FPGAs nutzt.
3. Nutzung und Erweiterung existierender Lösungen von Teilproblemen für die optimale Abbildung von Algorithmen auf heterogene Systeme und FPGA-Hardware.
4. Erforschung neuer (ggf. heuristischer) Methoden zur Optimierung von Programmen für inhärent parallele Architekturen.

Im Jahr 2018 wurden folgende wesentlichen Beiträge geleistet:

- Entwicklung eines source-to-source Übersetzerprototypen für die Umschreibung von OpenMP-C-Quellcode (vgl. Ziel 2).
- Entwicklung eines HLS-Übersetzerprototypen, der in der Lage ist, C-Code in Hardware zu übersetzen. Dieser Prototyp bildet die Basis für die Ziele 3 und 4.
- Entwicklung mehrerer experimenteller FPGA-Infrastrukturen für die Ausführung von Beschleunigerkernen (nötig für die Ziele 1 und 2).

Im Jahr 2019 wurden folgende wesentlichen Beiträge geleistet:

- Veröffentlichung zweier Papiere: „OpenMP on FPGAs - A Survey“ und „OpenMP to FPGA Offloading Prototype using OpenCL SDK“.
- Erweiterung des source-to-source Übersetzerprototypen um OpenMP-Target-Outlining (incl. Smoke-Tests).
- Fertigstellung des technischen Durchstichs für den ORKA-HPC-Prototypen (OpenMP-zu-FPGA-Übersetzer).
- Benchmark-Suite für die quantitative Leistungsanalyse von ORKA-HPC.
- Erweiterung des source-to-source Übersetzerprototypen um das Genom für die genetische Optimierung der High-Level-Synthese durch Einstellen von HLS-Pragmas.
- Prototypische Erweiterung des TaPaSCo-Composers um ein (optionales) automatisches Einfügen von Hardware-Synchronisationsprimitiven in TaPaSCo-Systeme.

Im Jahr 2020 wurden folgende wesentlichen Beiträge geleistet:

- Weiterentwicklung der Genetischen Optimierung.
- Aufbau eines Docker-Containers für zuverlässige Reproduzierbarkeit der Ergebnisse.
- Integration der Softwarekomponenten der Projektpartner.
- Plugin-Architektur für Low-Level-Platformen.
- Implementation und Integration zweier LLP-Plugin-Komponenten.
- Erweiterung des akzeptierten OpenMP-Sprachstandards.
- Erweiterung der Test-Suite.

Im Jahr 2021 wurden folgende wesentlichen Beiträge geleistet:

- Erweiterung der Benchmark-Suite.
- Erweiterung der Test-Infrastruktur.

- Erfolgreicher Projektabschluss mit Live-Demo für den Projektträger.
- Evaluation des Projekts.
- Veröffentlichung der Publikation „ORKA-HPC - Practical OpenMP for FPGAs“.
- Veröffentlichung des Quell-Codes und der Disseminationsumgebung auf Github.
- Erweiterung des akzeptierten OpenMP-Sprachstandards um neue OpenMP-Klauseln für die Steuerung der FPGA-bezogenen Transformationen.
- Weiterentwicklung der Genetischen Optimierung.
- Untersuchung des Verhältnisses von HLS-Leistungsschätzwerten und tatsächlichen Leistungskennzahlen.
- Aufbau eines linearen Regressionsmodells für die Vorhersage der tatsächlichen Leistungskennzahlen auf Basis der HLS-Schätzwerte.
- Entwicklung von Infrastruktur für die Übersetzung von OpenMP-Reduktionsklauseln.
- Erweiterung um die Übersetzung vom OpenMP-Pragma „parallel for“ in ein paralleles FPGA-System.

Im Jahr 2022 wurden folgende wesentlichen Beiträge geleistet:

- Generierung und Veröffentlichung eines Datensatzes zur Untersuchung des Verhältnisses von HLS-Ressourcenschätzwerten und tatsächlichen Leistungskennzahlen.
- Erstellung und vergleichende Evaluierung verschiedener Regressionsmodelle zur Vorhersage der tatsächlichen Systemperformanz aus frühen Schätzwerten.
- Analyse und Bewertung der durch die HLS generierten Ressourcenabschätzungen.
- Veröffentlichung der Publikation „Reducing OpenMP to FPGA Round-trip Times with Predictive Modelling“.
- Entwicklung eines auf dem Polyeder-Modell beruhenden Verfahrens zur Detektion und Entfernung von redundanten Lese-Operationen in FPGA-Stencil-Codes.
- Implementierung dieses Verfahrens in ORKA-HPC.
- Quantitative Evaluation der Stärken dieses Verfahrens und Ermittlung der Voraussetzungen, unter denen das Verfahren anwendbar ist.
- Veröffentlichung der Publikation „Employing Polyhedral Methods to Reduce Data Movement in FPGA Stencil Codes“.

3.5 SoftWater – *Software-Wasserzeichen*

Unter Software-Wasserzeichnen versteht man das Verstecken von ausgewählten Merkmalen in Programme, um sie entweder zu identifizieren oder zu authentifizieren. Das ist nützlich im Rahmen der Bekämpfung von Software-Piraterie, aber auch um die richtige Nutzung von Open-Source Projekten (wie zum Beispiel unter der GNU Lizenz stehende Projekte) zu überprüfen. Die bisherigen Ansätze gehen davon aus, dass das Wasserzeichen bei der Entwicklung des Codes hinzugefügt wird und benötigen somit das Verständnis und den Beitrag der Programmierer für den Einbettungsprozess. Ziel unseres Forschungsprojekts ist es, ein Wasserzeichen-Framework zu entwickeln, dessen Verfahren automatisiert beim Übersetzen des Programms Wasserzeichen sowohl in neu entwickelte als auch in bestehende Programme hinzufügen. Als ersten Ansatz untersuchten wir eine Wasserzeichenmethode, die auf einer symbolischen Ausführung und anschließender Funktionsynthese basiert.
Im Jahr 2018 wurden im Rahmen von zwei Bachelorarbeiten Methoden zur symbolischen Ausführung

und Funktionssynthese untersucht, um zu ermitteln, welche sich für unseren Ansatz am Besten eignet. Im Jahr 2019 wurde ein Ansatz auf Basis der LLVM Compiler Infrastruktur untersucht, der mittels konkolischer Ausführung (concolic execution, eine Kombination aus symbolischer und konkreter Ausführung) ein Wasserzeichen in einem ungenutzten Hardwareregister versteckt. Hierzu wurde der LLVM-Registerallokator dahingehend verändert, dass er ein Register für das Wasserzeichen freihält. Im Jahr 2020 wurde das inzwischen LLWM genannte Rahmenprogramm für das automatische Einfügen von Software-Wasserzeichen in Quellcode auf Basis der LLVM Compiler Infrastruktur um weitere dynamische Verfahren erweitert. Grundlage der hinzugefügten Verfahren sind, unter anderem, das Ersetzen/Verschleiern von Sprungadressen sowie Modifikationen des Aufrufgraphen. Im Jahr 2021 wurde das Rahmenprogramm LLWM um weitere angepasste, bereits in der Literatur bekannte, dynamische Verfahren sowie um das eigene Verfahren erweitert, das wir nun IR-Mark nennen Die hinzugefügten Verfahren basieren unter anderem auf der Umwandlung von bedingten Konstrukten in semantisch äquivalenten Schleifen oder auf Integrieren von Hashfunktionen, die die Funktionalität des Programms unverändert lassen, die Widerstandsfähigkeit aber erhöhen. IR-Mark wählt nun nicht nur gezielt die wenigen Funktionen aus, in denen die Registerverwendung bei der Code. Erzeugung verändert wird, sondern umfasst nun auch dynamische Aspekte um in den freigehaltenen Registern sinnvoll erscheinende Tarnwerte zu berechnen. Ein Artikel über LLWM und IR-Mark konnte publiziert werden. Im Jahr 2022 wurde das Rahmenprogramm LLWM um ein weiteres angepasstes Verfahren ergänzt. Die Methode nutzt Ausnahmebehandlungen, um das Wasserzeichen zu tarnen.

3.6 V&ViP – *Verifikation und Validierung in der industriellen Praxis*

Erkennung von Flaky-Tests auf Basis von Software-Versionsdaten und Testausführungshistorie

Regressionstests werden häufig und aufgrund ihres großen Umfangs zumeist vollautomatisiert ausgeführt. Sie sollen sicherstellen, dass Änderungen an einzelnen Komponenten eines Softwaresystems keine unerwünschten Nebenwirkungen auf das Verhalten von Teilsystemen haben, die von den Modifikationen eigentlich gar nicht betroffen sein sollten. Doch selbst wenn ein Testfall ausschließlich unveränderten Code ausführt, kann es trotzdem vorkommen, dass er manchmal erfolgreich ist und manchmal fehlschlägt. Derartige Tests nennt man „flaky“ und die Gründe dafür können sehr vielfältig sein, u.a. Wettlaufsituationen bei nebenläufiger Ausführung oder vorübergehend nicht verfügbare Ressourcen (z.B. Netzwerk oder Datenbanken). Flaky-Tests sind für den Testprozess in jeder Hinsicht ein Ärgernis, denn sie verlangsamen oder unterbrechen sogar die gesamte Testausführung und sie untergraben das Vertrauen in die Testergebnisse: Ist ein Testlauf erfolgreich, kann daraus nicht zwangsläufig geschlossen werden, dass das Programm diesbezüglich wirklich fehlerfrei ist, und schlägt der Test fehl, müssen ggf. teure Ressourcen investiert werden, um das Problem zu reproduzieren und ggf. zu beheben.
Der einfachste Weg, Test-Flakyness zu erkennen, besteht darin, Testfälle wiederholt auf der identischen Code-Basis auszuführen, bis sich das Testergebnis ändert oder mit einer gewissen statistischen Aussagesicherheit davon auszugehen ist, dass der Test nicht „flaky“ ist. Im industriellen Umfeld ist dieses Vorgehen jedoch selten möglich, da Integrations- oder Systemtests extrem zeit- und ressourcenaufwendig sein können, z.B. weil sie die Verfügbarkeit spezieller Test-Hardware voraussetzen. Aus diesem Grund ist es wünschenswert, die Klassifikation von Testfällen hinsichtlich Flakyness ohne wiederholte Neuausführung vorzunehmen, sondern dabei ausschließlich auf die bereits verfügbaren Informationen aus den bisherigen Entwicklungs- und Testphasen zurückzugreifen.
Im Jahr 2022 wurden verschiedene sogenannte Black-Box-Verfahren zur Erkennung von Test-Flakyness vergleichend untersucht, in einem realen industriellen Testprozess mit 200 Testfällen evaluiert und in ein praktisches Werkzeug implementiert. Die Klassifikation eines Testfalls erfolgt dabei ausschließlich auf Basis allgemein verfügbarer Informationen aus Versionskontrollsystemen und Testausführungswerkzeugen - also insbesondere ohne aufwändige Analyse der Codebasis oder Überwachung der Testüber-

deckung, die im Falle eingebetteter Systeme in den meisten Fällen ohnehin unmöglich wäre. Von den 122 verfügbaren Indikatoren (u.a. z.B. die Testausführungszeit, die Anzahl der Code-Zeilen oder die Anzahl der geänderten Code-Zeilen in den letzten 3, 14 und 54 Tagen) wurden verschiedene Teilmengen extrahiert und ihre Eignung für die Erkennung von Test-Flakyness bei Verwendung unterschiedlicher Verfahren untersucht. Zu diesen Verfahren zählen regelbasierte Methoden (z.B. „ein Test ist flaky, wenn er mind. fünfmal innerhalb des Beobachtungsfensters fehlgeschlagen ist, aber dabei nicht fünfmal hintereinander"), empirische Bewertungen (u.a. die Bestimmung der kumulierten gewichteten „flip rate", also die Häufigkeit des Alternierens zwischen Testerfolg und -fehlschlag) sowie verschiedene Verfahren aus der Domäne des Maschinellen Lernens (u.a. Klassifikationsbäume, Random Forest oder Multi-Layer Perceptrons). Die Verwendung KI-basierter Klassifikatoren zusammen mit dem SHAP-Ansatz zur Erklärbarkeit von KI-Modellen führte zur Bestimmung der wichtigsten vier Indikatoren („features") für die Bestimmung der Test-Flakyness im konkret untersuchten industriellen Umfeld. Als optimal hat sich dabei das sog. „Gradient Boosting" mit der kompletten Indikatorenmenge herausgestellt (F1-score von 96,5%). Nur marginal niedrigere Accuracy- und Recall-Kennwerte (bei nahezu gleichem F1-score) konnte das gleiche Verfahren mit nur vier ausgewählten Features erzielen.

Synergien von vor- und nachgelagerten Analysemethoden zur Erklärung künstlicher Intelligenz

Der Einsatz künstlicher Intelligenz verbreitet sich rasant und erobert immer neue Domänen des täglichen Lebens. Nicht selten treffen Maschinen dabei auch durchaus kritische Entscheidungen: Bremsen oder Ausweichen beim autonomen Fahren, Kredit(un)würdigkeit privater Personen bzw. von Unternehmen, Diagnose von Krankheiten aus diversen Untersuchungsergebnissen (z.B. Krebserkennung aus CT/MRT-Scans) u.v.m. Damit ein solches System im produktiven Einsatz Vertrauen verdient, muss sichergestellt und nachgewiesen sein, dass die gelernten Entscheidungsregeln korrekt sind und die Realität widerspiegeln. Das Trainieren eines maschinellen Modells selbst ist ein sehr ressourcenintensiver Prozess und die Güte des Ergebnisses ist in der Regel nur mit extrem hohem Aufwand und fundiertem Fachwissen nachträglich quantifizierbar. Der Erfolg und die Qualität des erlernten Modells hängt nicht nur von der Wahl des KI-Verfahrens ab, sondern wird im besonderen Maße vom Umfang und der Güte der Trainingsdaten beeinflusst.
Im Jahr 2022 wurde daher untersucht, welche qualitativen und quantitativen Eigenschaften eine Eingabemenge haben muss („a-priori-Bewertung"), um damit ein gutes KI-Modell zu erzielen („a-posteriori-Bewertung"). Dazu wurden verschiedene Bewertungskriterien aus der Literatur vergleichend bewertet und darauf aufbauend vier Basisindikatoren definiert: Repräsentativität, Redundanzfreiheit, Vollständigkeit und Korrektheit. Die zugehörigen Metriken erlauben eine quantitative Bewertung der Trainingsdaten im Vorfeld. Um die Auswirkung schlechter Trainingsdaten auf ein KI-Modell zu untersuchen, wurde mit dem sog. „dSprites"-Datensatz experimentiert, einem verbreiteten Generator für Bilddateien, der bei der Bewertung von Bilderkennungsverfahren eingesetzt wird. Damit wurden gezielt verschiedene Trainingsdatensätze generiert, die sich jeweils in genau einem der vier Basisindikatoren unterscheiden und dabei quantitativ unterschiedliche „a-priori-Güte" haben. Damit wurden jeweils zwei verschiedene KI-Modelle trainiert: Random Forest und Convolutional Neural Networks. Anschließend wurde die Güte der Klassifikation durch das jeweilige Modell anhand der üblichen statistischen Maße (Accuracy, Precision, Recall, F1-score) quantitativ bewertet. Zusätzlich wurde SHAP (ein Verfahren zur Erklärbarkeit von KI-Modellen) genutzt, um die Gründe für eine etwaige Missklassifikation bei schlechter Datenlage zu ermitteln. Wie erwartet, korreliert die Modellqualität mit der Trainigsdatenqualität: Je besser letztere hinsichtlich der vier Basisindikatoren abschneiden, desto genauer klassifiziert das trainierte Modell unbekannte Daten. Eine Besonderheit hat sich jedoch bei der Redundanzfreiheit herausgestellt: Erfolgt die Bewertung eines trainierten Modells mit komplett neuen/unbekannten Eingaben, dann ist die Genauigkeit der Klassifikation teils signifikant schlechter, als wenn die verfügbaren Eingabedaten in einen Trainings- und einen Evaluationsdatensatz geteilt werden: In letzteren Fall suggeriert die a-posteriori-Bewertung irreführend eine höhere Modellqualität.

Few-Shot Out-Of-Domain-Erkennung in der maschinellen Verarbeitung natürlicher Sprache

Die maschinelle Verarbeitung natürlicher Sprache („Natural Language Processing“, kurz NLP) hat viele Anwendungsgebiete, z.B. telefonische oder schriftliche Dialogsystemen (sog. Chat-Bots), die eine Kino-Auskunft erteilen, eine Eintrittskarte buchen, eine Krankmeldung aufnehmen oder Antworten auf diverse Fragen in bestimmten industriellen Abläufen geben. Häufig beteiligen sich derartige Chat-Bots auch in sozialen Medien, um z.B. kritische Äußerungen zu erkennen und ggf. zu moderieren. Mit zunehmendem Fortschritt auf dem Gebiet der künstlichen Intelligenz im Allgemeinen und der NLP im Speziellen, verbreiten sich zunehmend selbstlernende Modelle, die ihr fachliches und sprachliches Wissen erst während des konkreten praktischen Einsatzes dynamisch (und daher meist unüberwacht) ergänzen. Doch derartige Ansätze sind empfänglich für absichtlich oder unabsichtlich bösartige Beeinflussung. Beispiele aus der industriellen Praxis haben gezeigt, dass Chat-Bots schnell z.B. rassistische Äußerungen in sozialen Netzen „erlernen“ und anschließend gefährdende extremistische Äußerungen tätigen. Daher ist es von zentraler Bedeutung, dass NLP-basierte Modelle zwischen gültigen „In-Domain (ID)“ und ungültigen „Out-Of-Domain (OOD)“ Daten (also sowohl Ein- als auch Ausgaben) unterscheiden können. Dazu benötigen die Entwickler eines NLP-Systems für das initiale Training des KI-Modells jedoch eine immense Menge an ID- und OOD-Trainingsdaten. Während erstere schon schwer in hinreichender Menge zu finden sind, ist die a-priori-Wahl der letzteren i.d.R. kaum sinnvoll möglich.
Im Jahr 2022 wurden daher verschiedene Ansätze zur OOD-Erkennung untersucht und vergleichend bewertet, die mit wenigen bis keinen („few-shot“) Trainingsdaten funktionieren. Als Grundlage für die experimentelle Evaluierung diente das derzeit beste und am weitesten verbreitete, Transformer-basierte und vortrainierte Sprachmodell RoBERTa. Zur Verbesserung der OOD-Erkennung wurden u.a. „fine-tuning“ eingesetzt und untersucht, wie zuverlässig die Anpassung eines vortrainierten Modells an eine konkrete Domäne funktioniert. Zusätzlich wurden verschiedene Scoring-Verfahren implementiert und evaluiert, um Grenzwerte für die Klassifikation von ID- und OOD-Daten zu bestimmen. Um das Problem der fehlenden Trainingsdaten zu lösen, wurde auch ein Verfahren namens „data augmentation“ evaluiert: Dabei wurden mittels GPT3 („Generative Pretrained Transformer 3“, ein autoregressives Sprachmodell, das Deep Learning verwendet, um menschenähnlichen Text zu erzeugen) zusätzliche ID- und OOD-Daten für das Training bzw. die Evaluation von NLP-Modellen generiert.

4 Lehre

Der Lehrstuhl für Programmiersysteme bietet im Wintersemester das Pflichtmodul *Algorithmen und Datenstrukturen (AuD)* und im Sommersemester *Parallele und Funktionale Programmierung (PFP)* an. Da diese Module fakultätsübergreifend auch anderen Studiengängen (insbesondere Informations- und Kommunikationstechnik, Mathematik u.v.a.) angeboten werden, erreichten die Hörerzahlen mit 486 (AuD im WS2021/22) bzw. 413 (PFP im SS2022) im Berichtszeitraum erneut nahezu den Rekordwert der Vorsemester, die sich schließlich auch in der hohen Zahl an Prüfungsanmeldungen (421 in AuD bzw. 270 in PFP) niederschlagen. In der Vertiefungsrichtung Programmiersysteme bietet der Lehrstuhl verschiedene Module zu den Themen *Übersetzerbau*, *Clustercomputing* und *Testen von Softwaresystemen* an. Die Seminare *Hallo Welt! für Fortgeschrittene* und *Machine Learning* waren erneut innerhalb kürzester Zeit restlos ausgebucht.
Insgesamt betreute der Lehrstuhl für Programmiersysteme im Berichtsjahr drei Masterarbeiten und drei Bachelorarbeiten.

ICPC – *International Collegiate Programming Contest an der FAU*: Seit 1977 wird der International Collegiate Programming Contest (ICPC) ausgetragen. Dabei sollen Teams aus je drei Studierenden ca. 13 Programmieraufgaben lösen. Als Erschwernis kommt hinzu, dass nur ein Computer pro Gruppe zur

Verfügung steht. Die Aufgaben erfordern solide Kenntnisse von Algorithmen aus allen Gebieten der Informatik und Mathematik, wie z.B. Graphen, Kombinatorik, Zeichenketten, Algebra und Geometrie. Bei der Lösung kommt es darauf an, einen effizienten und richtigen Algorithmus zu finden und zu implementieren.
Der ICPC wird jedes Jahr in drei Stufen ausgetragen. Zuerst werden innerhalb der Universitäten in lokalen Ausscheidungen die maximal drei Teams bestimmt, die dann zu den regionalen Wettbewerben entsandt werden. Erlangen liegt seit dem Jahr 2009 im Einzugsbereich des Northwestern European Regional Contest (NWERC), an dem u.a. auch Teams aus Großbritannien, den Benelux-Staaten und Skandinavien teilnehmen. Die Sieger aller regionalen Wettbewerbe der Welt (und einige Zweitplatzierte) erreichen die World Finals, die im Frühjahr des jeweils darauffolgenden Jahres (2023 in Sharm El Sheikh, Egypten) stattfinden.
Am 29. Januar 2022 fand nach einer einjährigen Pause wieder der Winter Contest statt, dieses mal unter Organisation unserer Kollegen vom CPUlm, und erneut im Online-Format. Mitgemacht haben 59 Teams aus 12 Hochschulen und Universitäten, davon 6 Teams aus Erlangen. Unser bestes Team erreichte Platz 12. Am 27. Juni fand der German Collegiate Programming Contest an mehreren deutschen Universitäten und Hochschulen wieder in Präsenz statt, mit 5 Teams aus Erlangen. Das beste FAU-Team erreichte Platz 4 der 73 teilnehmenden Teams aus ganz Deutschland. Der NWERC fand am 27. November in Delft statt. Die FAU wurde durch 2 Teams vertreten, die die Plätze 43 und 60 bei 136 teilnehmenden Teams erreichten.Das Hauptseminar „Hallo Welt! - Programmieren für Fortgeschrittene“ hat im Jahr 2022 nicht stattgefunden.

5 Publikationen

[1] Mohammad Alawieh, Ernst Eberlein, Stephan Jaeckel, Norbert Franke, Birendra Ghimire, Tobias Feigl, George Yammine, and Christopher Mutschler. Complementary Semi-Deterministic Clusters for Realistic Statistical Channel Models for Positioning. In *Proc. IEEE Intl. Global Communications Conf. (GLOBECOM)*, pages 1–5, Rio de Janeiro, Brazil, December 2022. arXiv:2207.07837 [eess.SP]. doi:10.48550/arXiv.2207.07837.

[2] Mohammad Alawieh, George Yammine, Ernst Eberlein, Birendra Ghimire, Norbert Franke, Stephan Jaeckel, Tobias Feigl, and Christopher Mutschler. Towards Realistic Statistical Channel Models For Positioning: Evaluating the Impact of Early Clusters. In *Proc. IEEE Intl. Global Communications Conf. (GLOBECOM)*, pages 1–5, Rio de Janeiro, Brazil, December 2022. arXiv:2207.07838 [eess.SP]. doi:10.48550/ARXIV.2207.07838.

[3] Thorsten Blaß. *Ein datenparalleler Ansatz zur Beschleunigung von Datenflussanalysen mittels GPU*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, March 2022. doi:10.25593/978-3-96147-494-3.

[4] Julian Brandner, Florian Mayer, and Michael Philippsen. Dataset for: "Reducing OpenMP to FPGA Round-trip Times with Predictive Modelling". Zenodo, September 2022. doi:10.5281/zenodo.7534796.

[5] Julian Brandner, Florian Mayer, and Michael Philippsen. Reducing OpenMP to FPGA Round-trip Times with Predictive Modelling. In Michael Klemm, Bronis R. de Supinski, Jannis Klinkenberg, and Brandon Neth, editors, *OpenMP in a Modern World: From Multi-device Support to Meta Programming, Proc. 18th Intl. Workshop on OpenMP (IWOMP 2022)*, volume 13527 of *Springer's Lecture Notes in Computer Science (LNCS)*, pages 94–108, Chattanooga, TN, September 2022. Springer. doi:10.1007/978-3-031-15922-0_7.

[6] Tobias Brieger, Nisha, Lakshmana Raichur, Dorsaf Jdidi, Felix Ott, Tobias Feigl, Johannes Rossouw Van Der Merwe, Alexander Ruegamer, and Wolfgang Felber. Multimodal Learning for Reliable Interference Classification in GNSS Signals. In *Proc. Intl. Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+)*, pages 3210–3234, Denver, CO, September 2022. URL: https://www.ion.org/gnss/abstracts.cfm?paperID=11411, doi:10.33012/2022.18586.

[7] Veronika Dashuber and Michael Philippsen. Static And Dynamic Dependency Visualization in a Layered Software City. *SN Computer Science*, 3:Article 511, October 2022. doi:10.1007/s42979-022-01404-6.

[8] Veronika Dashuber and Michael Philippsen. Trace visualization within the Software City metaphor: Controlled experiments on program comprehension. *Information and Software Technology*, 150:Article 106989, June 2022. doi:10.1016/j.infsof.2022.106989.

[9] Jonathan Hansen, Johannes Rossouw Van Der Merwe, David Franco Contreras, Tobias Feigl, Tobias Brieger, Dorsaf Jdidi, Alexander Ruegamer, and Wolfgang Felber. Initial Results of a Low-Cost GNSS Interference Monitoring Network. In *Proc. Intl. Conf. on Positioning and Navigation for Intelligent Transport Systems (POSNAV)*, pages 1–10, Berlin, Germany, November 2022.

[10] Dorsaf Jdidi, Tobias Brieger, Felix Ott, Tobias Feigl, David Contreras Franco, Johannes Rossouw Van Der Merwe, Alexander Ruegamer, and Wolfgang Felber. Machine Learning Compression for GNSS Interference Analysis. In *Proc. Intl. Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+)*, Denver, CO, 2022. URL: https://www.ion.org/gnss/abstracts.cfm?paperID=11470.

[11] Dorsaf Jdidi, Tobias Brieger, David Tobias Feigl añd Contreras Franco, J. Rossouw van der Merwe, Alexandẽr Rügamer, Jochen Seitz, and Wolfgang Felber. Unsupervised Disentanglement for Post-Identificati on of GNSS Interference in the Wild. In *Proc. Intl. Technical Meeting of the Satellite Divis ion of The Institute of Navigation (ION GNSS+)*, pages 1176–1208, Denver, CO, 2022. doi:10.33012/2022.18493.

[12] Sebastian Kram, Christopher Kraus, Tobias Feigl, Maximilian Stahlke, Jörg Robert, and Christopher Mutschler. Position Tracking using Likelihood Modeling of Channel Features with Gaussian Processes, 2022. doi:10.48550/ARXIV.2203.13110.

[13] Sebastian Kram, Christopher Kraus, Maximilian Stahlke, Tobias Feigl, Jörn Thielecke, and Christopher Mutschler. Delay Estimation in Dense Multipath Environments using Time Series Segmentation. In *Proc. IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1671–1676, Austin, TX, April 2022. doi:10.1109/WCNC51071.2022.9771875.

[14] Florian Mayer, Julian Brandner, Matthias Hellmann, Jesko Schwarzer, and Michael Philippsen. The ORKA-HPC Compiler — Practical OpenMP for FPGAs. In Sunita Chandrasekaran Xiaoming Li, editor, *Proc. 34th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC 2021)*, volume 13181 of *Lecture Notes in Computer Science (LNCS)*, pages 83–97, Newark, DE, October 2022. Springer. URL: https://lcpc2021.github.io/pre_workshop_papers/Mayer_lcpc21.pdf, doi:10.1007/978-3-030-99372-6.

[15] Florian Mayer, Julian Brandner, and Michael Philippsen. Employing Polyhedral Methods to Reduce Data Movement in FPGA Stencil Codes. In *Prof. 35rd Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC 2022)*, Chicago, IL, October 2022.

[16] Nisha Lakshmana Raichur, Tobias Brieger, Dorsaf Jdidi, Carlo Schmitt, Birendra Ghimire, Felix Ott, Tobias Feigl, Johannes Rossouw Van Der Merwe, Alexander Ruegamer, and Wolfgang Felber.

Machine Learning-assisted GNSS Interference Monitoring through Crowd-sourcing. In *Proc. Intl. Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+)*, pages 1151–1175, Denver, CO, September 2022. URL: https://www.ion.org/gnss/abstracts.cfm?paperID=11470, doi:10.33012/2022.18492.

[17] Maximilian Stahlke, Tobias Feigl, Mario H. Castañeda García, Stirling-Gallacher Richard A., Jochen Seitz, and Christopher Mutschler. Transfer Learning to adapt 5G AI-based Fingerprint Localization across Environments. In *Proc. IEEE Vehicular Technology Conference (VTC-Spring)*, pages 1–5, Helsinki, Finland, June 2022. doi:10.1109/VTC2022-Spring54318.2022.9860906.

[18] Maximilian Stahlke, George Yammine, Tobias Feigl, Bjoern M. Eskofier, and Christopher Mutschler. Indoor Localization with Robust Global Channel Charting: A Time-Distance-Based Approach. *Proc. IEEE Transactions on Learning Technologies*, 1(1):1–12, 2022. doi:10.48550/arXiv.2210.06294.

[19] Johannes Rossouw Van Der Merwe, David Franco Contreras, Dorsaf Jdidi, Tobias Feigl, Alexander Ruegamer, and Wolfgang Felber. Low-cost COTS GNSS interference detection and classification platform: Initial results. In *Proc. Intl. Conf. on Localization and GNSS (ICL-GNSS)*, pages 1–8, Tampere, Finland, June 2022. doi:10.1109/ICL-GNSS54081.2022.9797025.