Software-Verifikation

5

Beiträge der Münchner Forschung

Dirk Beyer, Rolf Hennicker, Martin Hofmann, Tobias Nipkow und Martin Wirsing

Zusammenfassung

Bei der Entwicklung von Software ist es aufgrund ihrer Komplexität erforderlich, automatisierte Verfahren zur Überprüfung der korrekten Funktion einzusetzen. Um die Integration und korrekte Zusammenarbeit verschiedener Komponenten großer Softwaresysteme sicherzustellen, entwickeln Forschergruppen an den Münchner Universitäten sowohl Modell-basierte als auch Quellcode-basierte Techniken. Als Anwendungsbereiche betrachten wir unter anderem Komponenten der Infrastruktur, autonome Systeme und service-orientierte Systeme. Wir setzen Methoden und Algorithmen ein, die den Ingenieur während der Modellierung, der Programmierung und im Releasezyklus mit automatischen Techniken zur Abstraktion, Transformation und Verifikation unterstützen.

5.1 Einleitung

Korrekt funktionierende Software ist aus unserem täglichen Leben nicht mehr wegzudenken: alle Bereiche der Gesellschaft werden durchdrungen von Software und den dadurch bereitgestellten neuen Möglichkeiten zur Kommunikation, Informationsverarbeitung, Steuerung und Regelung. Für die Wirtschaft ist die Software ein treibender Technologiekatalysator aufgrund der ohne Software nicht erreichbaren Ausfallsicherheit von Systemen, Flexibitität bzgl. der Funktionen und Anforderungen, Kostenreduktion und schnellen Verfügbarkeit.

D. Beyer (✉) · R. Hennicker · M. Hofmann · M. Wirsing
Ludwig-Maximilians-Universität München
München, Deutschland

T. Nipkow
Technische Universität München
München, Deutschland

A. Bode et al. (Hrsg.), *50 Jahre Universitäts-Informatik in München*,
DOI 10.1007/978-3-662-54712-0_5

Das Ballungszentrum München zeichnet sich nicht nur als Standort leistungsstarker IT-Unternehmen aus, sondern auch durch ein großes Angebot an Lehre auf dem Gebiet der Software-Verifikation und durch die Präsenz einiger renommierter Forschergruppen im Bereich der Konstruktion und Verifikation von Software. Dieser Artikel soll eine kleine Auswahl an Themen aus dem Bereich der Software-Verifikation in München beschreiben; die Vorgehensweise wird exemplarisch anhand aktuell durchgeführter Projekte erläutert. Es gibt und gab an den Münchner Universitäten viele weitere Arbeiten und Arbeitsgruppen, die sich die Entwicklung korrekter Software mit formalen Methoden zum Ziel gesetzt und dazu wichtige Beiträge geleistet haben. Insbesondere sind hier die Pionierarbeiten im CIP-Projekt (Bauer) [2, 3], der strombasierte Ansatz (Broy) [17, 18], die Beiträge zum Model-Checking unendlicher Systeme (Esparza) [16, 22] und im weiteren Sinne auch die Beiträge der Universität der Bundeswehr auf dem Gebiet der modellbasierten Softwareentwicklung (Borghoff, Minas) und der IT-Sicherheit (Dreo, Hommel) zu nennen.

Exemplarisch für die Zusammenarbeit der Universitäten in diesem Bereich ist das gemeinsam an TUM und LMU München eingerichtete und von der DFG geförderte Graduiertenkolleg „Programm- und Modellanalyse“ (PUMA, 2008-2017). Dieser Verbund hat zum Ziel, „die wesentlichsten derzeit eingesetzten Methoden der Programm- und Modellanalyse zusammenzuführen und auf software-intensive Systeme anzuwenden“ [23]. Als diese Methoden wurden identifiziert: Programmverifikation durch Theorembeweisen, Model-Checking, statische Datenfluss-Analyse, sowie Typinferenz.

In all diesen Gebieten wurden in den letzten Jahren wegweisende Ergebnisse erzielt. So wurden die bisher mächtigste typbasierte Laufzeit- und Speicherplatzanalyse für funktionale Programme „RAML“, eines der weltweit erfolgreichsten Werkzeuge für die vollautomatische Verifikation von C-Programmen „CPAchecker“, einer der weltweit populärsten interaktiven Theorembeweiser „Isabelle“ und erstmals ein vollständig verifizierter Model-Checker für LTL „CAVA“ entwickelt. Weitere wichtige Beiträge gab es auf dem Gebiet der Strategie-Iteration, der Synthese reaktiver Systeme und dem Model-Checking probabilistischer und kontinuierlicher Systeme.

5.2 Ansätze und Algorithmen

Formale Entwicklungstechniken

Die Erforschung von Techniken, Methoden und Werkzeugen zur Entwicklung korrekter Software hat in München eine lange Tradition. Ausgangspunkt war in den 70er-Jahren des letzten Jahrhunderts das von F. L. Bauer und K. Samelson initiierte und geleitete CIP-Projekt zum „Computer-aided, Intuition-guided Programming“ [1] mit dem Ansatz, korrekte Programme aus formalen Spezifikationen mittels (von Hand) verifizierter Programmtransformationen herzuleiten. Dieser Ansatz wurde ab etwa 1990 an der TUM in der Gruppe von M. Broy und an der LMU in der Gruppe von M. Wirsing durch Kombination mit pragmatischen Softwareentwicklungstechniken und automatischen Verifikati-

onsverfahren weiterentwickelt. In diesem Abschnitt werden neuere Forschungsergebnisse skizziert, die die Verzahnung von Software-Engineering und Verifikationstechniken zur Entwicklung von komponentenbasierten, service-orientierten und autonomen Systemen illustrieren.

Im Rahmen des von der LMU federführend geleiteten EU-Projekts SENSORIA wurde eine Entwicklungsmethodik für service-orientierte Systeme erforscht [35, 36]. Dabei werden verwendet: eine service-orientierte Erweiterung von UML, Transformationen von Modellen dieser UML-Erweiterung in verschiedene Prozesskalküle und quantitative Analysemethoden mit der stochastischen Prozessalgebra PEPA. Zur Modellierung von Services werden Protokolle eingesetzt, die nötige und mögliche Aktionen von Services unterscheiden und mit Hilfe der MIO-Workbench verifiziert werden können [4].

Kollektive adaptive Systeme von autonom agierenden Komponenten waren Gegenstand der Forschung im EU Projekt ASCENS, das ebenfalls von der LMU Forschergruppe federführend geleitet wurde [37]. Beispiele für kollektive adaptive Systeme sind Roboterschwärme, dezentrale Peer-to-Peer-Cloud-Systeme oder Ensembles kooperierender Elektro-Automobile. Solche Systeme zeichnen sich durch ein hochgradig dynamisches Verhalten aus: Komponenten schließen sich „on-demand“ zu Ensembles zusammen, um bestimmte Aufgaben kollektiv zu lösen, einzelne Komponenten passen ihr Verhalten an Änderungen ihrer Umgebung an und ganze Architekturen können sich dynamisch ändern, indem Komponenten hinzukommen, wegfallen oder ausgetauscht werden. Die Forschergruppe an der LMU hat eine umfassende Entwicklungsmethodik für solche ensemblebasierten Systeme erarbeitet, die alle Schritte des Software-Lebenszyklusses umfasst [29].

Zusätzlich zu den klassischen „Design-Zeit“-Entwicklungsphasen wie Anforderungsanalyse, Modellierung, Programmierung,Validierung und Verifikation berücksichtigt der Software-Lebenszyklus für Ensembles auch Laufzeitaufgaben wie Monitoring, „Awareness“ und Selbstadaption. Software-Installation und -Update sowie das Feeedback des Systems verbinden Design-Zyklus und Laufzeit-Zyklus (siehe Abb. 5.1). Zur Modellierung und prototypischen Implementierung von Ensembles wurde die HELENA-Methodik entwickelt. HELENA bietet eine domänenspezifische grafische und textuelle Sprache zur Modellierung und Programmierung von Ensembles, sowie eine integrierte Workbench zur Codegenerierung und zur Verifikation [28].

Eine umfangreiche Anwendung eines komponenten-orientierten Systems wurde im Rahmen des Projekts GLOWA-Danube entwickelt, in dem bis zu 15 Simulationsmodelle verschiedener Forschergruppen aus Natur- und Sozialwissenschaften gekoppelt wurden, um integrative Simulationen zu Auswirkungen des globalen Klimawandels im Einzugsbereich der Oberen Donau durchzuführen [27]. Die Korrektheit der zeitlichen Koordination der einzelnen Simulationsmodelle wurde mit Methoden des Model-Checking nachgewiesen. Generell ist die Verifikation der korrekten Interaktion zwischen nebenläufig arbeitenden Komponenten schwierig, insbesondere, wenn die Komponenten in einer asynchronen Umgebung ausgeführt werden. Ein vielversprechender Ansatz besteht darin, ein synchrones Modell zu verifizieren und daraus Rückschlüsse auf die Eigenschaften einer asynchronen Implementierung zu ziehen [26].

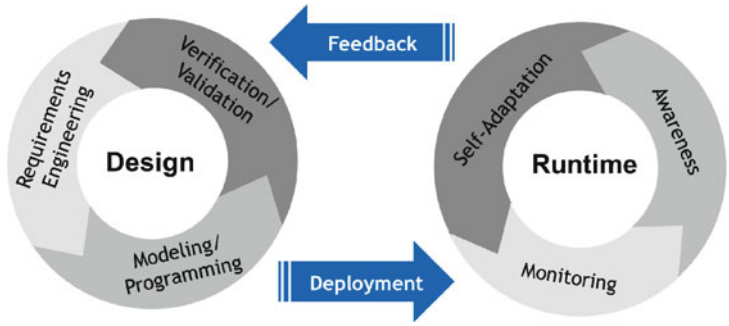

Abb. 5.1 Ensemble Development Life Cycle

Typanalyse

Ausgangspunkt der typbasierten Analyse sind die aus Java, C oder auch Haskell bekannten Typsysteme. Sie geben Auskunft über Anzahl und Formate der Parameter und des Resultats einer Methode, sowie in begrenztem Maße über ihre Seiteneffekte. Darüber hinaus werden die Typannotierungen in Sprachen wie Java nach einfachen und für den Programmierer nachvollziehbaren Regeln propagiert und verifiziert.

Man betrachte zum Beispiel die folgende Typisierung einer Methode zum Auffinden der Matrikelnummer einer Person an einer Universität.

```
MatNr findMN(Person p, Univ u) throws NotFoundException
```

Diese Typisierung stellt u. a. sicher, dass das erste Argument die gesuchte Person und das zweite die Universität bezeichnet und nicht umgekehrt. Außerdem wird klar, dass das Ergebnis eine Matrikelnummer ist und dass möglicherweise eine bestimmte Exception ausgelöst wird. Man vergleiche die Situation mit der folgenden schwächeren Typisierung, wie man sie in C und ähnlichen Sprachen findet.

```
int findMN(char **p, char *u)
```

Bei vollkommen untypisierten Sprachen wie LISP ist nicht einmal die Anzahl der Argumente statisch am Programmtext ablesbar, geschweige denn ihr Format.

Die angekündigten Typen aller im Programm verwendeten Methoden müssen tatsächlich stimmen, in dem Sinne, dass die entsprechenden Methodenrümpfe den angekündigten Typ erfüllen, also Ergebnisse des angekündigten Typs liefern und höchstens die angekündigten Seiteneffekte auslösen. Etwaige Methodenaufrufe, auch rekursive, dürfen hierbei bereits als korrekt vorausgesetzt werden: liefert man typkorrekte Parameter, so kann man davon ausgehen, dass das Resultat den angekündigten Typ hat und höchstens die angekündigten Seiteneffekte ausgelöst werden.

Die typbasierte Analyse verallgemeinert nun diese Typen auf feinere Eigenschaften von Daten und feinere Beschreibungen der möglichen Seiteneffekte und deren zeitlicher Abfolge. Als konkretes Beispiel wollen wir hier eine typbasierte Version einer *taintedness*-Analyse betrachten, die gegen Angriffe wie SQL-Injektion und Cross-Site-Scripting helfen kann. Beide bilden nach wie vor neben Phishing die wichtigsten Angriffe auf webbasierte IT-Systeme.

Eine schöne Illustration bietet der Web-Comic https://xkcd.com/327/, in dem Eltern ihrem Jungen den eigenwilligen Vornamen

```
Robert'); DROP TABLE Students;--
```

gegeben haben. Wird dieser Vorname in ein Web-Formular eingegeben und von der darunterliegenden Software ohne weitere Vorkehrungen in eine SQL-Abfrage eingesetzt, so führt das dazu, dass server-seitig die Tabelle `Students` gelöscht wird. Als Abhilfe wird empfohlen, von auswärtigen Benutzern gelieferte Strings niemals ungeprüft in SQL-Abfragen, HTML-Seiten, JavaScript-Befehlen und ähnliches einzubauen, sondern sie immer mit geeigneten Framework-Methoden zu *sanitisieren*.

Mit typbasierter Analyse kann man nun sicherstellen, dass diese Sanitisierung tatsächlich stattfindet. In der einfachsten Form würde man hier den Typ `String` in zwei Subtypen `String@ok` und `String@user` aufteilen. Benutzereingaben, die von einer geeigneten Framework-Methode wie `getParameter()` bereitgestellt werden, bekommen immer den Typ `String@user`. Rückgabewerte einer Sanitisierungsmethode, sowie String-Literale aus dem Programmtext erhalten den Typ `String@ok`. Die Framework-Methode, die SQL-Anfragen aus Strings präpariert, verlangt nun den Typ `String@ok`.

Diese verfeinerten Typen erscheinen natürlich auch an anderen Stellen, an denen in einem Programm Typen auftreten, insbesondere werden Klassen je nach den verfeinerten Typen ihrer Felder und Methoden in Unterklassen aufgeteilt werden, wodurch unter dem Stichwort „regionenbasierte Analyse“ [5] eine sehr genaue Berücksichtigung von Aliasing-Effekten erfolgen kann.

Die Typisierung von Feldern illustriert einen weiteren Aspekt der typbasierten Analyse: Regionen. Hat man zum Beispiel eine Klasse `StringBuf` mit einem Feld *s* des Typs `String`, so kann man diese Klasse auf zwei Arten verfeinern: `StringBuf@a` und `StringBuf@b`, je nachdem, ob das Feld `s` ein `String@ok` oder ein `String@user` ist. Die Annotate `a` und `b` heißen *Regionen*. Jedes Objekt einer bestimmten Klasse gehört zu einer Region; die Felder und Methoden unterschiedlicher Objekte können unterschiedlich verfeinerte Typen haben.

Hat man ein Objekt `o` der Klasse `StringBuf@a`, dann wäre es vielleicht verlockend, seinen Typ nach einer Zuweisung der Form

```
o.s = getParameter();
```

auf `StringBuf@b` zu „verschlechtern“. Das ist aber in Gegenwart von Aliasing inkorrekt: hat man vorher z. B. `o` an eine Methode, die ein `StringBuf@a` erwartet, über

```
1  public void doGet(HttpServletRequest request) {
2    String input = request.getParameter();
3
4    // case 1: HTML embedding
5    String s = "<body>" + escapeToHtml(input) + "</body>";
6    output(s);
7
8    // case 2: JavaScript embedding
9    if (showAlert) {
10     output("<script>");
11     output("  alert('" + escapeToJs(input) + "');");
12     output("</script>");
13   }
14 }
```

Abb. 5.2 Beispiel für die Verwendung eines Sanitisierungs-Frameworks

den formalen Parameter *x* des Types `StringBuf@a` übergeben, so wird im Rumpf der Methode erwartet, dass `x.s` ein sanitisierter String ist, was nach der genannten Zuweisung nicht mehr garantiert werden kann.

Es ist möglich, solche als *Typestate* bekannte Typveränderungen nach Zuweisung korrekt zu implementieren, das erfordert aber eine sog. lineare Typisierung, welche die Einrichtung von Aliases stark beschränkt. Für das hier beschriebene Szenario ist das nicht sinnvoll; es gibt aber z. B. Anwendungen im Rahmen der typbasierten Analyse von State-Pattern in modernen Sprachen mit linearer Typisierung wie Rust.

In vielen Anwendungen muss man die Strings noch weiter verfeinern. So haben wir eine von SAP-Research an uns herangetragene Fallstudie beschrieben [24], in der durch sequentielle String-Ausgaben sukzessive eine Web-Seite zusammengestellt wird. Je nach Art des bisher ausgegebenen Kontexts müssen Benutzereingaben unterschiedliche Sanitisierungsfunktionen durchlaufen; im konkreten Fall sind das vier verschiedene, jeweils für JavaScript, URLs, HTML und SQL, siehe Beispiel in Abb. 5.2. Die verfeinerten String-Typen geben dann präzise Auskunft über die Natur eines Kontexts und die durchlaufenen Sanitisierungsfunktionen. Die entsprechenden Annotierungen und Typisierungsregeln können aus einem endlichen Policy-Automaten (Abb. 5.3) generiert werden. Nachdem Stringausgaben in dieser Fallstudie über Seiteneffekte und verteilt erfolgen, werden hier außerdem Seiteneffekt-Annotierungen eingeführt, welche ganz analog zu den `throws`-Klauseln beschreiben, welcher Art die bisher ausgegebene Zeichenkette ist.

Im aktuellen Projekt GuideForce [21] entwickeln wir eine effiziente Implementierung, welche Typ-Annotationen nicht nur prüft, sondern auch automatisch berechnet. Darüber hinaus kann das System konfiguriert werden und so auch auf andere für die sichere Web-Programmierung entwickelte Richtlinien angepasst werden. Die Genauigkeit geschlossener kommerzieller Systeme wie CheckMarx, Coverity, u. ä. kann so erreicht oder übertroffen werden bei gleichzeitig kompletter Transparenz der zugrundeliegenden Regeln.

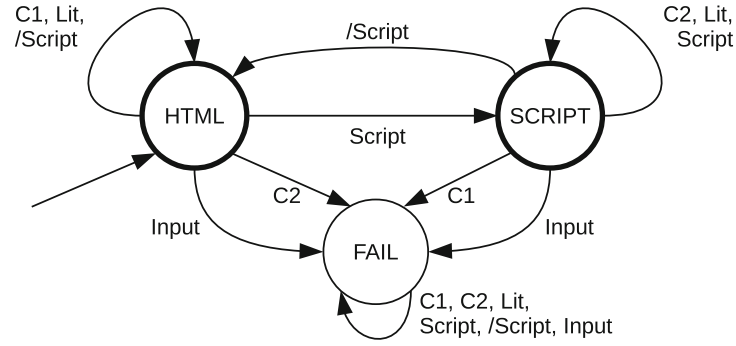

Abb. 5.3 Vereinfachter Policy-Automat

Datenfluss-Analyse und Software-Model-Checking

Um trotz der enormen Komplexität der zu verifizierenden Softwaresysteme vollautomatische und präzise Verifikation von Sicherheitseigenschaften betreiben zu können, müssen Techniken aus verschiedenen Bereichen zusammenfließen. Es reicht weder aus, klassische Ansätze aus dem Bereich des Compilerbaus wie Lattice-basierte Datenflussanalyse [30] zu verwenden, weil diese zwar hocheffizient, aber zu ungenau sind, noch ist es zielführend, erschöpfende Beweistechniken wie Model-Checking [19] zu verwenden, weil diese zwar genau genug, jedoch leider ein viel zu ressourcen-intensives Laufzeitverhalten haben. Moderne Verfahren kombinieren die Vorzüge der beiden Ansätze [10]; eine erfolgreiche Umsetzung ist die Configurable-Program-Analysis (CPA) [11], die es bequem ermöglicht, Kombinationen von Datenfluss-Analyse und Model-Checking zu beschreiben und zu implementieren. So zum Beispiel wird der Ansatz der Gegenbeispiel-gesteuerten Abstraktionsverfeinerung (CEGAR) [20] genutzt für neue Kombinationen mit klassischen Analysen wie Value-Analysis [13] und Symbolic-Execution [12].

Ein ernst zu nehmendes aktuelles Problem der automatischen Verifikation ist es, dass der Einsatz immer noch sehr viel Expertenwissen über das eingesetzte Werkzeug verlangt. Die aktuellen Ansätze sind in Forschungswerkzeugen implementiert, aber noch nicht ausgereift genug für den Standardentwickler. Außerdem ist es nicht zufriedenstellend, wie die Kommunikation zwischen Verifikationswerkzeug und Benutzer erfolgt. Traditionell gibt ein Verifikationswerkzeug als Antwort TRUE oder FALSE, was für den Anwender keinesfalls ausreichend ist. An der LMU werden daher Verfahren und Formate entwickelt und weiterentwickelt, die dieses Problem beheben: Werkzeug-unabhängige und austauschbare Formate für sogenannte Verifikationszeugen schaffen die Möglichkeit zur Speicherung wertvoller, zusätzlicher Informationen als Beiprodukt des Verifikationsvorganges. Bisher bewährt haben sich der Violation-Witness [9] und der Correctness-Witness [8]. Der Violation-Witness wird für den Fall der Ausgabe FALSE produziert und mit dem Verifikationsresultat abgelegt. Er enthält Hinweise, die es einem unabhängigen Validierer

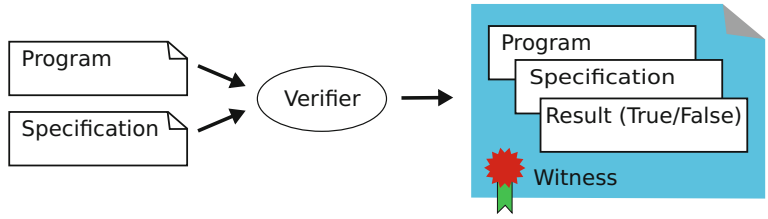

Abb. 5.4 Witness als Resultat des Verifikationsprozesses

ermöglichen, den gefundenen Fehler zu rekonstruieren. Daraus können teilweise Testfälle abgeleitet und mittels traditioneller Methoden untersucht werden [7]. Der Correctness-Witness speichert Invarianten und andere Hinweise, die einem Benutzer oder einem Validator hilfreich sind zu verstehen, warum das Programm die Sicherheitseigenschaft erfüllt. Insbesondere für Regressionsanalyse ist es wichtig, bei erneuten Verifikationsläufen Aufwand zu sparen, indem vorherige Resultate wiederverwendet werden [14].

Abb. 5.4 illustriert den *neuen* Verifikationsprozess mit Witness: Das Programm und die formale Spezifikation der Sicherheitseigenschaft werden dem Verifizierungswerkzeug als Eingabe übergeben. Als Ausgabe wird dem Benutzer nicht nur TRUE oder FALSE zurückgemeldet, sondern ein Witness, der Hinweise dafür enthält, dass die Verifikation des gegebenen Programmes und der gegebenen Spezifikation das Verifikationsresultat ergibt und reproduzierbar sein sollte. Ein unabhängiger Validierer kann nun basierend auf dem Witness, aber unabhängig von dem für die Verifikation benutzten Werkzeug, den Nachweis liefern, ob das Verifikationsresultat stimmt. Das Verfahren mit der Möglichkeit der unabhängigen Validierung erhöht das Vertrauen in den Gesamtprozess und wird erfolgreich in Verifikationswettbewerben zur Qualitätsverbesserung eingesetzt [6].

Theorembeweisen

Seit 1992 wird der interaktive Theorembeweiser Isabelle [33, 34] an der TUM in Zusammenarbeit mit Lawrence Paulson von der Universität Cambridge und mit Markus Wenzel entwickelt (mit langjähriger Förderung durch die DFG). Isabelle ist einer der beiden weltweit populärsten Theorembeweiser. Mit seiner Hilfe kann man Beweise beliebig komplexer formaler Aussagen interaktiv erstellen, deren Korrektheit vom System überprüft wird. Dies reicht von der Software-Verifikation bis hin zur Mathematik. Zu den herausragenden Ergebnissen zählt die Verifikation des Java-Bytecode-Verifiers [31] (GI-Dissertationspreis 2003), die Entwicklung des ersten verifizierten Betriebssystemkerns [32], und ein signifikanter Beitrag zum verifizierten Beweis der Keplerschen Vermutung [25] (zur dichtesten Packung von gleichen Kugeln).

5.3 Werkzeuge und praktische Nutzbarkeit

Freie Verfügbarkeit Eine Gemeinsamkeit der oben erwähnten Forschungsprojekte an den Münchner Universitäten ist es, die Forschungsergebnisse in Werkzeugen öffentlich zur Verfügung zu stellen. Die an den Lehrstühlen entwickelten Softwaresysteme werden der Gesellschaft und der Wirtschaft durch Open-Source-Lizenzen zur freien Verfügung gestellt. Dadurch kann die Software weltweit eingesetzt werden und auch von externen Forschungsteams weiterentwickelt werden. So zum Beispiel entstand Isabelle unter Mitarbeit von über 100 Entwicklern weltweit; für das CPAchecker-Projekt werden 29 Entwickler gelistet, die innerhalb der letzten 12 Monate mehr als 3500 Änderungen (Commits) eingereicht haben[1]

Artifakt-Sammlungen Das *Archive of Formal Proofs*[2], eine ständig wachsende Online-Bibliothek von Isabelle-Beweisen, umfasste im Januar 2017 insgesamt 330 Beiträge mit 1,5 Mio. Zeilen von 243 Autoren. Die *Collection of Verification Tasks*[3] ist eine Sammlung von Verifikationsproblemen, die in einem GitHub-Projekt öffentlich verfügbar sind und allgemein zur experimentellen Evaluation zur Verfügung stehen. Aktuell befinden sich im Repository mehr als 15.000 Programme in den Programmiersprachen C und Java im Umfang von 122 Mio. Zeilen Quelltext (3,7 GB).

5.4 Methoden

Die Berechnungen zur Analyse der Korrektheit von Computerprogrammen sind extrem rechenaufwändig, und die meisten unserer Forschungsresultate müssen experimentell bestätigt werden.

Experimentelle Forschung

Experimentiert wird in der Softwaretechnik schon lange; um allerdings wissenschaftlich valide Ergebnisse aus umfangreichen Experimenten zu erhalten, müssen die Grundsätze des genauen Messens und der Darstellung von experimentellen Ergebnissen beachtet werden. So ist es z. B. erst seit einigen Jahren Dank einer neuen Funktionalität im Linux-Kernel möglich, den Verbrauch von CPU-Zeit und Hauptspeicher genau zu messen und Ressourcen-Grenzen zuverlässig zu forcieren. Dazu wurden neben der hauptsächlichen Algorithmenforschung Beiträge zur Definition von replizierbaren Experimenten geleistet. So zum Beispiel wird mit dem an der LMU weiterentwickelten Werkzeug benchexec die

[1] https://www.openhub.net/p/cpachecker.
[2] https://www.isa-afp.org/.
[3] https://github.com/sosy-lab/sv-benchmarks.

erste Experimentierplattform für die zuverlässige Ressourcen-Messung CPU-intensiver Berechnungsprozesse verfügbar [15].

Wettbewerbe

Eine besondere Art der Evaluierung von Forschungsresultaten, die in den letzten Jahren immer mehr an Bedeutung gewonnen hat, ist die Form des internationalen Werkzeug-Wettbewerbs. Solche Wettbewerbe existieren z. B. im Bereich „Satisfiability Modulo Theory“ (SMT-COMP) und im Bereich „Software Verification“ (SV-COMP). Entscheidungsprogramme wie SMT-Solvers oder Software-Verifiers werden automatisiert auf hunderte oder tausende von Eingabe-Problemen ausgeführt, und anhand der Korrektheit und Antwortzeit wird der Erfolg der Teilnehmer ermittelt und in Rankings präsentiert.

Die „International Competition on Software Verification“ (SV-COMP) [6] wird im Jahr 2017 an der LMU ausgerichtet. Durch den Rechencluster der Gruppe Beyer können dem Wettbewerb Ressourcen von über 1500 Rechenkernen und 5 TB Hauptspeicher zur Durchführung bereitgestellt werden. Umgerechnet auf einen normalen Desktop-PC beträgt die Rechenzeit ca. 3 Jahre, muss aber für den Wettbewerb innerhalb einiger Tage absolviert werden.

5.5 Ausblick

Dieser Artikel ist anlässlich des Jubiläums „50 Jahre Informatik in München“ entstanden. Er soll dazu dienen, einen groben Überblick zu geben, wie die Forschergruppen der Münchner Universitäten Theorien, Methoden und Werkzeuge beitragen, um korrekte Softwaresysteme zu entwickeln. Dazu wurden exemplarisch einige Themen vorgestellt, die den beteiligten Autoren am Herzen liegen, ohne einen Anspruch auf Vollständigkeit zu legen. Es ist geplant, diese Themen in der Zukunft weiter auszubauen, um weiterhin wertvolle Beiträge zur Konstruktion und Qualitätssicherung zu leisten.

Literatur

1. F. L. Bauer. Program development by stepwise transformations – The project CIP. Appendix: Programming languages under educational and under professional aspects. In *Program Construction, International Summer School, Marktoberdorf*, LNCS 69, pages 237–272. Springer, 1978. DOI: 10.1007/BFb0014671
2. F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*. LNCS 183. Springer, 1985. DOI: 10.1007/3-540-15187-7

3. F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations – Computer-aided, Intuition-guided Programming. *IEEE Trans. Software Eng.*, 15(2):165–180, 1989. DOI: 10.1109/32.21743
4. S. S. Bauer, P. Mayer, A. Schroeder, and R. Hennicker. On weak modal compatibility, refinement, and the MIO Workbench. In *Proc. TACAS*, LNCS 6015, pages 175–189. Springer, 2010.
5. L. Beringer, R. Grabowski, and M. Hofmann. Verifying pointer and string analyses with region type systems. *Computer Languages, Systems & Structures*, 39(2):49–65, 2013. DOI: 10.1016/j.cl.2013.01.001
6. D. Beyer. Software verification with validation of results (Report on SV-COMP 2017). In *Proc. TACAS*. Springer, 2017. LNCS 10206, pages 331–349, DOI: 10.1007/978-3-662-54580-5_20
7. D. Beyer and M. Dangl. Verification-aided debugging: An interactive web-service for exploring error witnesses. In *Proc. CAV (2)*, LNCS 9780, pages 502–509. Springer, 2016. DOI: 10.1007/978-3-319-41540-6_28
8. D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness witnesses: Exchanging verification results between verifiers. In *Proc. FSE*, pages 326–337. ACM, 2016. DOI: 10.1145/2950290.2950351
9. D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proc. FSE*, pages 721–733. ACM, 2015. DOI: 10.1145/2786805.2786867
10. D. Beyer, S. Gulwani, and D. Schmidt. Combining model checking and data-flow analysis. In E. M. Clarke, T. A. Henzinger, and H. Veith, editors, *Handbook on Model Checking*. Springer, 2017.
11. D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007. DOI: 10.1007/978-3-540-73368-3_51
12. D. Beyer and T. Lemberger. Symbolic execution with CEGAR. In *Proc. ISoLA*, LNCS 9952, pages 195–211. Springer, 2016. DOI: 10.1007/978-3-319-47166-2_14
13. D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013. DOI: 10.1007/978-3-642-37057-1_11
14. D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. Precision reuse for efficient regression verification. In *Proc. ESEC/FSE*, pages 389–399. ACM, 2013. DOI: 10.1145/2491411.2491429
15. D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. In *Proc. SPIN*, LNCS 9232, pages 160–178. Springer, 2015. DOI: 10.1007/978-3-319-23404-5_12
16. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. CONCUR*, LNCS 1243, pages 135–150. Springer, 1997.
17. M. Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, 3(1):21–57, 1991.
18. M. Broy and G. Ştefănescu. The algebra of stream processing functions. *Theoretical Computer Science*, 258(1):99–129, 2001.
19. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. Logic of Programs 1981*, LNCS 131, pages 52–71. Springer, 1982.
20. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
21. S. Erbatur and M. Hofmann. GuideForce: Type-based enforcement of programming guidelines. In *Proc. SEFM*, LNCS 9509, pages 75–89. Springer, 2015. DOI: 10.1007/978-3-662-49224-6_8

22. J. Esparza, P. Ganty, and T. Poch. Pattern-based verification for multithreaded programs. *ACM Trans. Program. Lang. Syst.*, 36(3):9, 2014.
23. J. Esparza, M. Hofmann, T. Nipkow, H. Seidl, DFG Graduiertenkolleg GRK 1480: *Programm und Modellanalyse (PUMA)*, 2008-2017, 2007.
24. R. Grabowski, M. Hofmann, and K. Li. Type-based enforcement of secure programming guidelines – Code injection prevention at SAP. In *Proc. FAST*, LNCS 7140, pages 182–197. Springer, 2011. DOI: 10.1007/978-3-642-29420-4_12
25. T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete and Computational Geometry*, 44:1–34, 2010.
26. R. Hennicker, M. Bidoit, and T.-S. Dang. On synchronous and asynchronous compatibility of communicating components. In *Proc. COORDINATION*, LNCS 9686, pages 138–156. Springer, 2016.
27. R. Hennicker, S. Janisch, A. Kraus, and M. Ludwig. A web-based modelling and decision support system to investigate global change and the hydrological cycle in the Upper Danube basin. In *Regional Assessment of Global Change Impacts – The Project GLOWA-Danube*, chapter 2, pages 19–28. Springer, 2016.
28. R. Hennicker, A. Klarl, and M. Wirsing. Model-checking Helena ensembles with Spin. In *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, LNCS 9200, pages 331–360. Springer, 2015.
29. M. M. Hölzl, N. Koch, M. Puviani, M. Wirsing, and F. Zambonelli. The ensemble development life cycle and best practices for collective autonomic systems. In *Software Engineering for Collective Autonomic Systems – The ASCENS Approach*, LNCS 8998, pages 325–354. Springer, 2015. DOI: 10.1007/978-3-319-16310-9_9
30. G. A. Kildall. A unified approach to global program optimization. In *Proc. POPL*, pages 194–206. ACM, 1973. DOI: 10.1145/512927.512945
31. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood. *Verified Java-Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.
32. G. Klein et al. seL4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.
33. T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. http://concrete-semantics.org. DOI: 10.1007/978-3-319-10542-0
34. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002. DOI: 10.1007/3-540-45949-9
35. M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, A. Knapp, N. Koch, and A. Schroeder. Semantic-based development of service-oriented systems. In *Proc. FORTE*, LNCS 4229, pages 24–45. Springer, 2006.
36. M. Wirsing and M. M. Hölzl, editors. *Rigorous Software Engineering for Service-Oriented Systems – Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*. LNCS 6582. Springer, 2011. DOI: 10.1007/978-3-642-20401-2
37. M. Wirsing, M. M. Hölzl, N. Koch, and P. Mayer, editors. *Software Engineering for Collective Autonomic Systems – The ASCENS Approach*. LNCS 8998. Springer, 2015. DOI: 10.1007/978-3-319-16310-9