Dr. Kiran Lakhotia

(E-Mail: k.lakhotia@cs.ucl.ac.uk)
arbeitet als wissenschaftlicher Mitarbeiter im CREST Zentrum des University College London. Sein Interesse liegt in der automatisierten Testdatengenerierung mit dem Schwerpunkt „Suchbasiertes Testen". Seine Arbeit umfasst allerdings auch statische Analysen und die Kombination von suchbasierten Testmethoden mit symbolischer Ausführung.

AUSTIN – automatisierte Strukturtests für C

Trotz regem akademischen Interesse und ansteigender Publikationen im Bereich der suchbasierten Testdatengenerierung existieren relativ wenige öffentliche Werkzeuge [Ton04], die automatisiertes suchbasiertes Testen umsetzen. AUSTIN, ein prototypisches Open-Source-Werkzeug für Modultests von C-Programmen, versucht diese Lücke zu füllen.

Ein suchbasierter Software-Test (SBT) formuliert die Suche nach Testfällen für ein Testkriterium als Optimierungsaufgabe. Dem unterliegt die Annahme, dass es einfacher ist, stichprobenartig (oder manuell) generierte Testdaten zu optimieren, anstatt auf Anhieb Testfälle zu erzeugen, die ein bestimmtes Testkriterium erfüllen.

In einem Optimierungsverfahren muss eine Zielfunktion definiert werden, die einen Suchalgorithmus zum globalen Optimum leitet. Im Rahmen der Testdatenerzeugung entspricht ein globales Optimum der Erfüllung eines Testkriteriums. Der Suchraum stellt die Menge aller Programmparameter der zu testenden Software dar. Eine Vielzahl von (Such-) Algorithmen eignen sich für die Optimierung einer Zielfunktion [Har09]. Am häufigsten, besonders bei Strukturtests, werden allerdings evolutionäre Algorithmen eingesetzt [McM04].

Bei Forschern ist SBT unter anderem wegen der Flexibilität von Suchalgoritmen beliebt. Es bedarf nur einer neuen Zielfunktion, um SBT in verschiedenen Bereichen wie Funktionstests, Mutationstests und Regressionstests einzusetzen. Trotz der zahlreichen Anwendungsmöglichkeiten für SBT konzentriert sich der Großteil der Forschung jedoch weiterhin auf Strukturtests. Dabei geht es speziell um die Erzeugung von Testdaten zur Zweigüberdeckung in einem Kontrollflussgraph (KFG).

Suchbasierte Testdatenerzeugung

Es gibt verschiedene Strategien, um Testdaten zur Zweigüberdeckung eines KFG zu generieren. Dabei unterscheidet man prinzipiell zwischen überdeckungsorientierten und zielorientierten Methoden. AUSTIN verwendet eine zielorientierte Methode, in der jeder Zweig des KFG ein separates Optimierungsproblem darstellt. Dabei spielt es keine Rolle, welchen Kontrollflusspfad die generierten Testdaten letztendlich ausführen, solange sie die jeweiligen Zielzweige erreichen.

Die Zielfunktion in AUSTIN [Aus], die den Optimierungsprozess leitet, entspricht dem Stand der Technik und besteht aus zwei Komponenten: einer Zweigdistanz (engl. *branch distance*) und einer Annäherungsstufe (engl. *approach level*). Sind beide Werte Null, ist das gesuchte Testdatum gefunden.

Zweige eines Kontrollflussgraphen sind oft von mindestens einem Verzweigungsknoten kontrollflussabhängig. Die Annäherungsstufe ist ein Maß dafür, wie viele Verzweigungsknoten, von denen ein Zweig abhängig ist, nicht von einem Testdatum ausgeführt wurden. Umso weniger dieser Verzweigungsknoten ein Testdatum erreicht, desto weiter, im Sinne eines Kontrollflussgraphen, ist es von einem Zielzweig entfernt.

Angenommen, das Ziel ist die Verzweigung zwischen Knoten 3 und 4 in der Beispielfunktion in Abbildung 1. Der Zielzweig ist von den Knoten 1, 2 und 3 kontrollflussabhängig. Wenn ein Testdatum dem ‚falsch'-Zweig des Knoten 1 folgt, ist die Annäherungsstufe 2, wenn es dem ‚falsch'-Zweig des Knoten 2 folgt, ist sie 1 und so weiter.

Node ID	Example
	void testme(int x, int y, int z)
	{
1	if(x == 0)
2	if(y == z)
3	if(x == z)
4	printf(„target");
	}

Abb. 1: Beispiel-Funktion

Wann immer ein Testdatum einen Zielzweig verfehlt, wird eine Zweigdistanz an jenem Verzweigungsknoten berechnet, an dem das Testdatum von einem Pfad zum Zielzweig abwich. Eine Zweigdistanz basiert auf der Verzweigungsbedingung eines Knoten und dient als Indiz dafür, wie weit ein Testdatum davon entfernt ist, eine bestimmte Bedingung zu erfüllen.

Folgt ein Testdatum zum Beispiel dem ‚falsch'-Zweig des Knoten 1 in Abbildung 1 so wird die Zweigdistanz anhand der Formel: $x - 0 + K$ (K ist eine Konstante, in diesem Falle 1) berechnet. Je geringer der

Abstand zu Null, umso näher ist ein Testdatum daran, den ‚wahr'-Zweig von Knoten 1 auszuführen. Für jede Art von Verzweigungsbedingung gibt es eine entsprechende Zweigdistanzformel (siehe Tracey et al. [Tra98]).

Man kann die Zweigdistanz als Minimierung der Annäherungsstufen betrachten und somit sollte eine Zielfunktion der Annäherungsstufe mehr Gewicht verleihen. Die komplette Zielfunktion besteht aus der Formel:

F = approach level + normalize (branch distance)

Die Funktion „normalize" ordnet der Zweigdistanz einen Wert aus dem Intervall [0,1] zu und ist in der Literatur entweder als $\frac{x}{x-1}$ oder $1 - 1.001^{-x}$ definiert, wobei x den Wert der Zweigdistanz darstellt.

Da die Zielfunktion in AUSTIN keinen Constraint Solver [Mou08] nutzt, kann die Generierung von Testdaten unter Umständen robuster sein als bei alternativen Methoden (wie z. B. Dynamische Symbolische Ausführung [God05]). Dies trifft insbesondere für Verzweigungsbedingungen zu, die Gleitkommazahlen beinhalten oder nicht linear sind.

AUSTIN verfügt über drei Suchalgorithmen um Parameterwerte zu erzeugen: eine Stichprobensuche (engl. *Random Search*), eine Abwechselnde Variablen Methode (AVM) (engl. *Alternating Variable Method*) und eine, durch symbolische Ausführung unterstützte AVM.

Random Search

Die Stichprobensuche weist allen Programmparametern einen Zufallswert innerhalb eines erlaubten Wertebereichs zu. Komplexe Datentypen wie Zeiger oder dynamische Datenstrukturen werden wie folgt initialisiert:

Für jeden Zeiger entscheidet ein Münzwurf zwischen einem Nullzeiger oder einer Speicheradresse. Im Fall der Speicheradresse wählt erneut ein Zufallsverfahren zwischen einer neuen Heap-Adresse (per *malloc*) oder einer bereits bestehenden Adresse, z. B. die eines anderen Programmparameters, aus. Dies ermöglicht es selbstreferenzierende Datenstrukturen zu erzeugen. Wann immer ein Zeiger per malloc auf eine neue Adresse zeigt, wiederholt sich der Initialisierungsvorgang für den entsprechenden Datentyp, der die Adresse enthält (z. B. die Felder in einer Datenstruktur).

Die Stichprobensuche benötigt keine Zielfunktion und nutzt nur eine einfache Instrumentierung des Quellcodes, um die erreichte Zweigüberdeckung auszuwerten. Falls ein Testdatum einen bis dato noch nicht ausgeführten Zweig, oder gar den Zielzweig ausführt, wird es in ein Archiv als Testfall übernommen. Ansonsten generiert AUSTIN solange neue Testdaten, bis es eine vollständige Zweigüberdeckung erreicht oder ein Stoppkriterium erfüllt. Stoppkriterien in metaheuristischen Optimierungsverfahren stellen sicher, dass eine Suche aufhört, selbst wenn sie kein globales Optimum findet. Das Stoppkriterium in AUSTIN legt fest, wie viele Testdaten maximal per Zielzweig generiert werden.

Alternating Variable Method

Die AVM ist ein lokaler Suchalgorithmus (ähnlich wie ein Hill Climber [McM04]), dessen Effektivität und Effizienz bereits mehrere Studien nachgewiesen haben [Lak10-a, Har10]. Die Suche startet mit einem stichprobenartig generierten Testdatum. Danach werden alle Programmparameter von einem primitiven Datentyp (integer, floats) in einem Vektor arrangiert. Falls das Testdatum den Zielzweig verfehlt, durchläuft die AVM jenen Vektor und unternimmt für jedes Element in Isolation so genannte „Exploratory Moves". Diese addieren (subtrahieren) ein kleines Delta vom Wert eines Parameters. Für ganzzahlige Datentypen fängt das Delta bei 1 an, d. h. mit dem kleinstmöglichen Zuwachs (Dekrement). Wenn eine Addition (Subtraktion) sich dem globalen Optimum der Zielfunktion nähert, versucht die AVM die Suche zu beschleunigen, indem sie das Delta bei jedem weiteren Schritt erhöht. Dies sind sogenannte „Pattern Moves". Der Wert des Deltas wird anhand folgender Formel berechnet:

In dieser Formel ist *it* die Anzahl der wiederholten Schritte bei einem „Pattern Move" (1 für „Exploratory Moves"), *dir* entweder -1 oder 1, und prec die Präzision des derzeitig manipulierten Programmparameters. Die Präzision ist nur für Gleitkommazahlen relevant (d. h. Null für Integer) und bestimmt wie grobkörnig eine Suche für solche Variablen ist. Legt man den Wert z. B. auf 1, addiert (subtrahiert) ein „Exploratory Move" minimal 0.1 von dem Wert eines Programmparameters. Bei einer Präzision von 2 liegt der kleinsten Wert des Deltas bei 0.01 und so weiter. Im Standard-Modus liegt AUSTINs Präzision bei 2. Ein Nutzer hat aber natürlich die Möglichkeit diesen Wert zu verändern.

Symbolically Enhanced Alternating Variable Method

Wie bereits geschildert, kann die Testdatengenerierung für dynamische Datenstrukturen und Zeiger in der Stichprobensuche oder AVM sehr ineffizient sein. Zeiger haben eine hohe Wahrscheinlichkeit, initialisiert zu werden, was oft zu großen Datenstrukturen führt. Deshalb verfügt AUSTIN über eine, durch symbolische Ausführung unterstützte, AVM. Das Ziel dabei ist es, Zeiger nur wenn nötig zu initialisieren, d. h. standardmäßig gelten alle Zeiger als Nullzeiger.

Um jedoch einen Zielzweig auszuführen, ist es oft erforderlich, dass Zeigern bestimmte Speicheradressen zugewiesen werden. Die Zweigdistanz, und damit die Zielfunktion, ist nicht dafür geeignet nutzvolle Informationen (d. h. einen Gradient) von konkreten Speicheradressen zu berechnen. Deshalb erstellt AUSTIN mithilfe einer symbolischen Ausführung Constraints zu den Zeigern, die dann von einem speziellen Constraint Solver gelöst werden.

Wann immer ein Testdatum einen Zielzweig verfehlt, führt AUSTIN jenen Kontrollflusspfad symbolisch aus, dem das Testdatum folgte. Das Ergebnis ist eine (symbolische) Pfadbedingung (engl. *Path Condition*) [Kin76], die aus einer Kombination von algebraischen Ausdrücken und konditionalen Operatoren besteht. Sie beschreibt, welche Bedingungen die Programmparameter eines Software Moduls erfüllen müssen, um einen bestimmten (Kontrollfluss-)Pfad durch das Modul auszuführen.

Für AUSTIN sind nur diejenigen Constraints interessant, die Zeigerparameter beinhalten. Alle anderen Constraints reflektieren Kontrollflussbedingungen, an denen eine Zweigdistanz verwendet werden kann. Für diese verwendet AUSTIN nach wie vor einen Suchalgorithmus. Aus den übrigen Constraints formt AUSTIN eine *Sub-Path Condition*, indem es alle Constraints über Programmparameter von primitiven

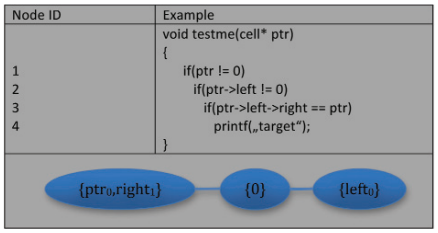

Abb. 2: Beispiel zur Erläuterung, wie Zeiger in AUSTIN gehandhabt werden.

Obfuscated Function Names	Lines Of Code	Number of Branches
Adaptive headlight control:		
02	919	420
03	259	142
06	58	36
Door lock control:		
07	85	110
08	99	76
11	199	129
Electric window control:		
12	67	32
15	272	216
Total	**1,958**	**1,161**

Tabelle 1: Übersicht von Funktionen aus dem Automobilbereich auf die AUSTIN angewandt wurde [Lak10-a]. Die „Lines Of Code" Tabellenspalte gibt die gesamte vorverarbeitete Länge des C-Quellcodes einer Funktion an. Sie wurde mit der Standardkonfiguration des CCCC Werkzeugs [Lit01] berechnet. Funktionsnamen sind obfuskiert.

Datentypen entfernt. Dazu gehören auch Constraints, in denen ein Zeiger zu einem primitiven Datentyp de-referenziert wird. Zudem ignoriert AUSTIN alle Constraints, die nicht von einem für den derzeitigen Zielzweig kontrollflussabhängigen Knoten stammen. Es bleiben daher nur Constraints über Zeiger (Speicheradressen) übrig. AUSTIN nutzt die CIL [Nec02] Werkzeugkette, damit alle verbleibenden Constraints dem Format x = y oder x ≠ y folgen, wobei x und y entweder einen Zeigerparameter oder die Nullkonstante darstellen.

Um sukzessive Ausführungen des Programms näher an einen Zielzweig heranzuführen, muss der letzte binäre Operand (d. h. ‚=' oder ‚≠') in der Sub-Path Condition umgedreht werden. Dies ergibt eine neue Path Condition, aus der AUSTIN einen ungerichteten Äquivalenzgraphen erstellt. Die Knoten des Graphen sind Äquivalenzklassen symbolischer Variablen. Die Äquivalenzbeziehung zwischen symbolischen Variablen wird durch ‚=' Operanden in der Path Condition ermittelt. Kanten zwischen den Knoten stellen Ungleichheiten (‚≠') dar. Der Äquivalenzgraph wird inkrementell aufgebaut und dient zur Lösung der Constraints über Zeigerparameter.

Zunächst prüft AUSTIN, ob alle Constraints der Path Condition erfüllbar sind. Ein Constraint ist nicht erfüllbar, wenn alle symbolischen Variablen zu demselben Knoten gehören und der binäre Operand in dem Constraint eine Ungleichheit (‚≠') darstellt. Gleichermaßen, wenn der binäre Operand eines Constraints eine Gleichheit (‚=') darstellt und die symbolischen Variablen des Constraints in zwei verschiedenen, durch eine Kante verbundenen Knoten liegen, ist der Constraint unerfüllbar.

Für jeden erfüllbaren Constraint erneuert AUSTIN den Äquivalenzgraph, indem es Knoten oder Kanten hinzufügt oder unverbundene Knoten zusammenlegt. **Abbildung 2** zeigt ein Beispielprogramm (oben) und wie der zugehörige Äquivalenzgraph (unten) aussieht, nachdem AUSTIN ein Testdatum generiert hat, welches Knoten 4 ausführt.

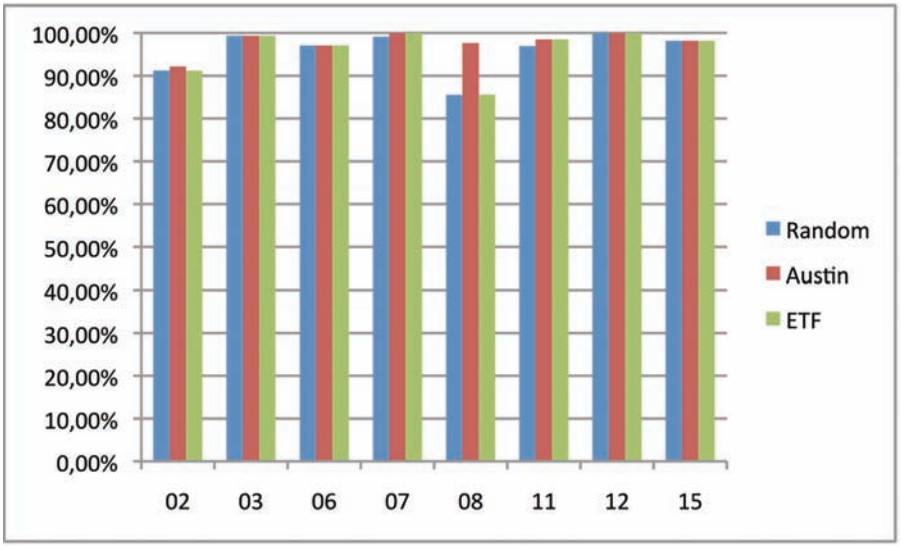

Abb. 3: Zweigüberdeckung von AUSTIN, ETF und einer Stichprobensuche für die Funktionen aus Tabelle 1.

Der AUSTIN Prototyp

AUSTIN, ein Kommandozeilenwerkzeug, basiert auf der „C Intermediate Language" (CIL) [Nec02] Werkzeugkette. CIL ermöglicht das Parsen von ANSI-C Quellcode und unterstützt Microsoft C und GNU C Erweiterungen. Zusätzlich bietet CIL verschiedene Repräsentationen des Quellcodes an, z. B. einen abstrakten Syntaxbaum und Kontrollflussgraph. Beide können ohne weiteres durch CIL's API manipuliert werden.

Während des Parsen von Quellcode wandelt CIL außerdem den Code in eine einfachere, gut strukturierte C-Variante um. Dabei ist eine Transformation von besonderer Bedeutung. Durch logische UND (&&) oder logische ODER (||) Operatoren zusammengesetzte Prädikate werden mit Hilfe von Code-Replikationen und „goto"-Konstruktionen in einzelne Bedingungen umgewandelt. Dies kann man zwar unterbinden, doch AUSTIN verwendet den Standardmodus von CIL, in dem diese Vereinfachung des Quellcodes vorgenommen wird. Als Folge dienen die von AUSTIN erzeugten Testdaten zur Bedingungs-/Entscheidungsüberdeckung (engl. *C/DC*) im originalen Quellcode, was einer Zweigüberdeckung im CIL-transformierten Code entspricht.

Wie oft in einem Prototyp üblich, gibt es in AUSTIN gewisse Limitationen. AUSTIN kann zum Beispiel keine Funktionen testen, die eine variable Anzahl von Parametern deklarieren (d. h. „variable length functions"). Zudem hat AUSTIN mit bestimmten Parametertypen Probleme. Parameter, die als Zeiger auf eine Funktion deklariert sind („function pointers"), werden immer als Nullzeiger gehandhabt, da AUSTIN für solche Zeiger keine sinnvollen Werte erzeugen kann. Dasselbe gilt für Zeiger auf „void", denn AUSTIN fehlen Analysen dafür, wie solche Zeiger genutzt werden und was die „wahren" Datentypen sind.

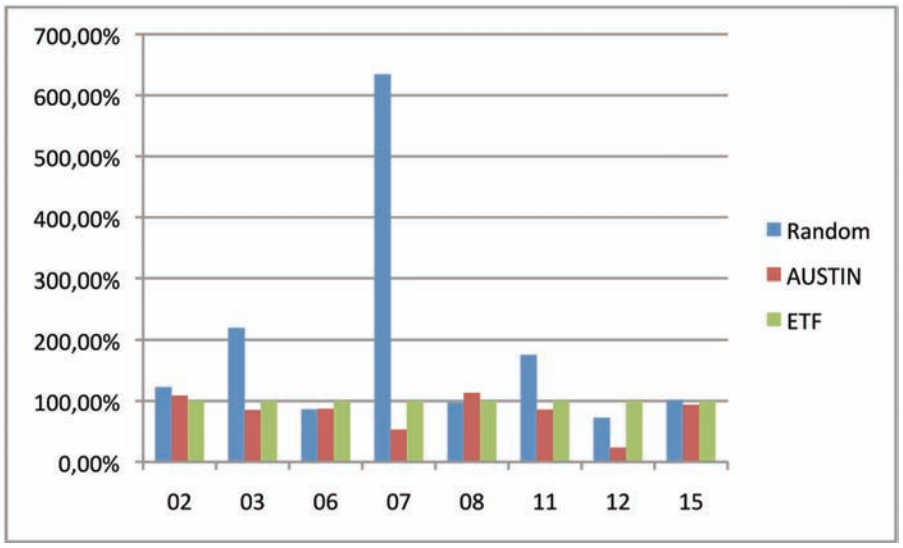

Abb. 4: Anzahl der benötigten Fitness-Auswertungen für die Funktionen aus Tabelle 1. Die Anzahl der Fitness-Auswertungen ist im Bezug auf das ETF normalisiert.

Ähnlich steht es um Programmparameter vom „union" Typ. Bei einer Union werden alle Mitglieder initialisiert, egal welches Mitglied letztendlich relevant ist. Dies hat zur Folge, dass das zuletzt initialisierte Mitglied einer Union alle vorhergehenden Werte überschreibt. Schließlich hat AUSTIN Schwierigkeiten, Testdaten für Funktionen mit einem String-Parametertyp zu generieren. In der C-Programmiersprache gibt es keinen expliziten String Datentyp, sondern Strings werden als char-Zeiger deklariert. Als solches werden sie auch von AUSTIN gehandhabt, d. h. AUSTIN weist ihnen entweder die Nullkonstante oder einen einzelnen char-Wert zu. Dies führt bei string-manipulierenden Funktionen wie „strcpy" oft zu Fehlern. Es ist zwar möglich, in AUSTIN einen char-Zeiger als Array zu markieren, um somit eine String-Eingabe zu simulieren, ein Nutzer ist jedoch selbst dafür verantwortlich, gegebenenfalls ein abschließendes Nullzeichen in solch ein Array einzufügen.

Trotz den aufgeführten Limitationen wurde AUSTIN bereits erfolgreich in mehreren Studien „out-of-the-box" eingesetzt. Dabei wurden teils große real-world [Lak10-a] Funktionen aus dem Automobilbereich, sowie zahlreiche Open-Source-Programme getestet. In beiden Fällen erreicht AUSTIN eine hohe Zweigüberdeckung und ist mit anderen Methoden wie der „Dynamic Symbolic Execution" oder dem „Evolutionary Testing Framework" (ETF) vergleichbar [Lak10-a, Lak10-b]. Das ETF wurde zum Beispiel von EvoTest [Evo06], einem

Referenzen

[AUS] AUSTIN, URL: http://code.google.com/p/austin-sbst/

[Ber] Berner & Mattner Systemtechnik GmbH, URL: http://www.berner-mattner.com/en/berner-mattner-home/company/index4l5.html.

[Evo06] EvoTest – Evolutionary Testing for Complex Systems, IST-33472, URL: http://evotest.iti.upv.es/.

[Gro09] H. Gross, P. M. Kruse, J. Wegener, and T. Vos, „Evolutionary white-box software test with the evotest framework: A progress report", in ICSTW '09, pages 111 – 120, 2009.

[God05] P. Godefroid, N. Klarlund and K. Sen, „DART: directed automated random testing", ACM SIGPLAN Notices, Vol. 40, No. 6, pages 213 – 223, 2005.

[Har09] M. Harman, A. Mansouri and Y. Zhang, „Search based software engineering: A comprehensive analysis and review of trends techniques and applications", Department of Computer Science, King's College London, Tech. Rep. TR-09-03, 2009.

[Har10] M. Harman and P. McMinn, „A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search", IEEE Transactions on Software Engineering, Vol. 36, No. 2, pages 226 – 247, 2010.

[Kin76] J. C. King, „Symbolic Execution and Program Testing", Communications of the ACM, Vol. 19, No. 7, pages 385 – 394, 1976.

[Lak10-a] K. Lakhotia, M. Harman, H. Gross, „AUSTIN: A tool for Search Based Software Testing for the C Language and its Evaluation on Deployed Automotive Systems", in SSBSE, pages 101 – 110, 2010.

[Lak10-b] K. Lakhotia, P. McMinn and M. Harman, „An Empirical Investigation Into Branch Coverage for C Programs Using CUTE and AUSTIN", Journal of Systems and Software (JSS) , Vol. 83, No. 12, pages 2379 – 2391, 2010.

[Lit01] T. Littlefair, „An Investigation Into The Use Of Software Code Metrics In The Industrial Software Development Environment", Ph.D. dissertation, Faculty of computing, health and science, Edith Cowan University, Australia, 2001.

[McM04] P. McMinn, „Search-based software test data generation: A survey", Software Testing, Verification and Reliability, Vol. 14, No. 2, pages 105 – 156, 2004.

[Mou08] Leonardo de Moura and Nikolaj Bjørner, „Z3: An Efficient SMT Solver", Tools and Algorithms for the Construction and Analysis (TACAS), Lecture Notes in Computer Science, Vol. 4963, pages 337 – 340, 2008.

[Nec02] G. C. Necula, S. McPeak, S.P. Rahul and W. Weimer, „CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs", in Proceedings of Conference on Compilier Construction, 2002.

[Ton04] P. Tonella, „Evolutionary Testing Of Classes", in ISSTA, pages 119 – 128, 2004.

[Tra98] N. Tracey, J. A. Clark, K. Mander, and J. A. McDermid, „An automated framework for structural test-data generation", in ASE, pages 285 – 288, 1998.

Test Subject	Number of Functions Tested	Number of Branches Tested	Lines of Code
defroster	2	76	250
f2	1	44	305
gimp	28	292	867
spice	2	142	269
Total	**33**	**554**	**1,691**

Tabelle 2: *Übersicht von Funktionen aus dem Automobilbereich und Open-Source-Programmen, auf die AUSTIN angewandt wurde. Wie in Tabelle 1 gibt die „Lines Of Code" Tabellenspalte die gesamte vorverarbeitete Länge des C-Quellcodes einer Funktion an.*

Europäischen Projekt entwickelt und wird in mehreren Firmen, mitunter auch bei Daimler, eingesetzt.

Tabelle 1 lassen sich acht Funktionen aus dem Automobilbereich entnehmen, die von Berner & Mattner Systemtechnik GmbH [Ber] für eine Studie im Rahmen des EvoTest-Projektes ausgewählt wurden [Gro09]. Die Studie verglich das ETF mit einer Stichprobensuche im Bezug auf die erreichte Zweigüberdeckung der Funktionen. Beide Methoden hatten das gleiche Fitness Budget zur Verfügung. Ein Fitness Budget legt fest, wie oft ein Programm im Rahmen einer Suche ausgeführt werden darf.

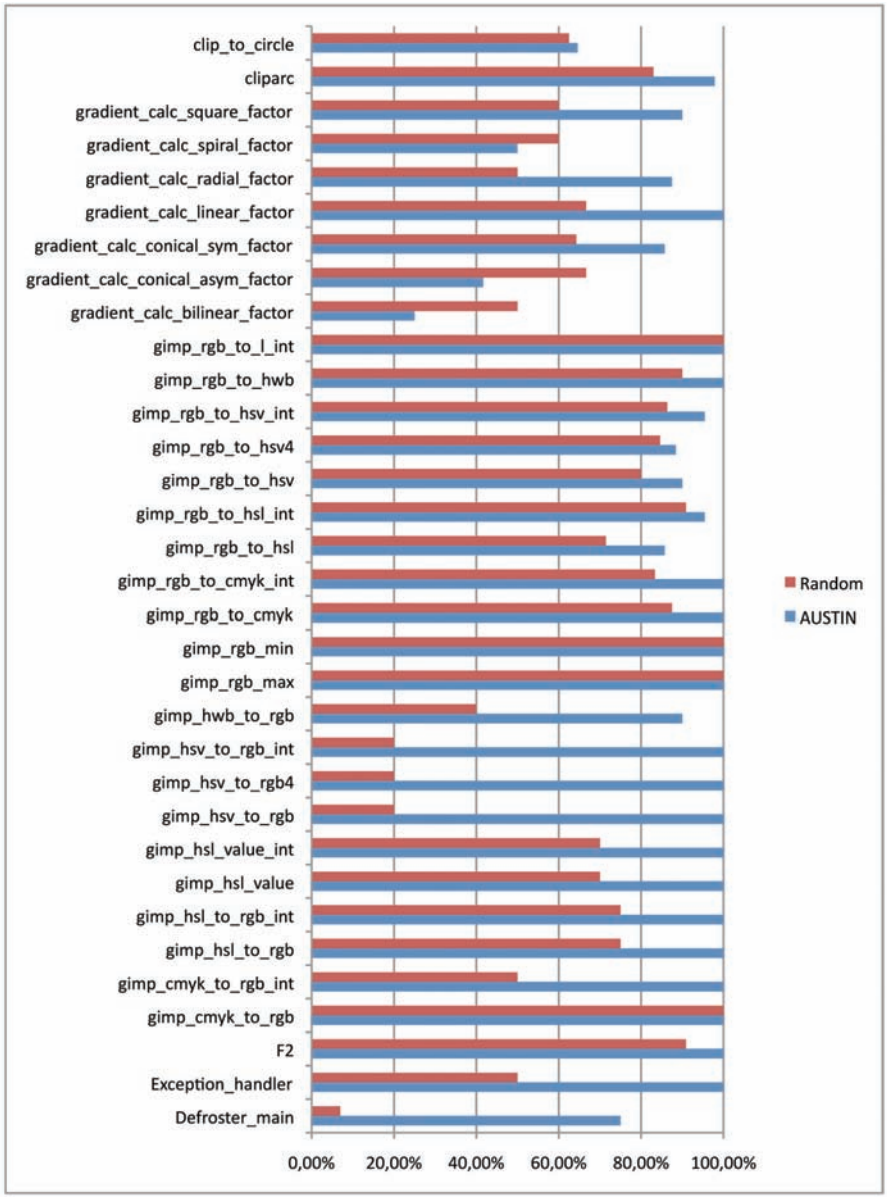

Abb. 5: *Zweigüberdeckung von AUSTIN und einer Stichprobensuche für die Funktionen aus Tabelle 2.*

Dieselben Funktionen wie in Tabelle 1 wurden ebenfalls mit AUSTIN und der „Symbolically Enhanced AVM" getestet. Abbildung 3 zeigt das Maß an Zweigüberdeckung (y-Achse) im Vergleich zu dem ETF und der Stichprobensuche. Das Ergebnis deutet darauf hin, dass AUSTIN mindestens ebenso effektiv ist wie das ETF, und bei zwei Funktionen sogar eine höhere Zweigüberdeckung als das ETF erreicht. Generell benötigt AUSTIN allerdings im Vergleich zu dem ETF und der Stichprobensuche weit weniger Fitnessauswertungen. Je weniger Fitnessauswertungen eine Suche benötigt desto effizienter, und somit schneller, ist sie. Abbildung 4 zeigt die (im Bezug auf das ETF) normalisierten Fitness-Auswertungen (y-Achse) für die 8 Funktionen. Eine Ausnahme bilden die Funktionen 02 und 08. Dort benötigt AUSTIN mehr Fitness-Auswertungen als das ETF. Allerdings erzielt AUSTIN am Ende für diese Funktionen auch eine höhere Zweigüberdeckung, was unter Umständen wichtiger sein kann als eine schnellere Testdatengenerierungsmethode.

Zusätzlich zu den Funktionen in Tabelle 1 wurde AUSTIN's „Symbolically Enhanced AVM" auch bei den Funktionen in Tabelle 2 angewandt. Die beiden Funktionen „defroster" und „f2" wurden von Daimler zur Verfügung gestellt. Defroster ist eine Funktion zur Kontrolle der Heckscheibenheizung und f2 trägt zur Motorensteuerung bei. Der Quellcode für beide Funktionen wurde automatisch von einem Modell erzeugt. Die Programme „gimp" und „spice" sind Open-Source. Gimp ist ein bekanntes Graphikprogramm und spice ist ein analoger Schaltkreissimulator. Abbildung 5 zeigt die Zweigüberdeckung von AUSTIN's „Symbolically Enhanced AVM" im Vergleich zu einer Stichprobensuche. Wie bereits in der vorherigen Studie, erzielt AUSTIN auch hier eine höhere Zweigüberdeckung als eine Stichprobensuche.

Danksagung

Vielen Dank an Lena Hierl, Dr. Jens Krinke und Prof. Dr. Ing. Ina Schieferdecker für das Korrekturlesen.