Integrierte Testkonzepte — Von der Theorie zur Praxis und zurück

Ralf Gerlich

ralf.gerlich@bsse.biz

Abstract: Die theoretische Forschungsarbeit hat bereits eine ganze Reihe von Verfahren und Strategien zur Automation von Teilen des Softwaretest hervorgebracht. Der Erfolg in der Praxis hängt aber wesentlich davon ab, diese Verfahren in entsprechenden Werkzeugen zur Anwendung verfügbar zu machen. Dabei ist ein integriertes Konzept notwendig, das den automatisierten bzw. teil-automatischen Test von in der Praxis vorkommenden Softwaresystemen ermöglicht. Ein solches Konzept wird durch die Betrachtung bereits existierender Werkzeuge wie auch die Vorstellung weiterführender Ansätze präsentiert. Sich hieraus ergebende neue Problemstellungen für Theorie und Forschung werden diskutiert.

1 Einführung

Der klassische Softwaretest hat die Entdeckung von Fehlern im Prüfling zum Ziel. Dabei wird der Prüfling mit Testeingaben stimuliert, die Ausgaben des Prüflings werden registriert und mit den erwarteten Ausgaben verglichen. Aus dieser Aufteilung ergeben sich die folgenden drei Abschnitte eines Softwaretests:

- Testdatenerzeugung,
- Testausführung (Stimulation), und
- Testauswertung.

Während in der Grundlagenforschung die Freiheit besteht, sich mit spezifischen Fragestellungen zu jedem dieser Themen einzeln zu beschäftigen, werden für die Praxis Werkzeuge und Methoden benötigt, die die Anwendung auf im industriellen Alltag vorkommende Softwaresysteme ermöglicht. Vereinfachende Annahmen aus der Theorie sind dann oftmals nicht mehr zulässig.

Auch das wirtschaftliche Umfeld in Form von verfügbaren Ressourcen spielt eine große Rolle. Dieser Einfluss geht in Teilen so weit, dass Anforderungen an die Testdurchführung, die eigentlich nur Minimalanforderungen sein sollten, zum absoluten Testziel auserkoren werden. Äußerst schwache Abdeckungskriterien werden plötzlich zum einzig wahren Bewertungsmaßstab für Testqualität.

Im Jahr 1992, in dem die Entwicklung des Zufallstestwerkzeugs DARTT[GF93] (Dynamic Ada Random Test Tool) begonnen wurde, war etwa das Interesse an solchen Werkzeugen

bei den Qualitätssicherungsverantwortlichen so gut wie nicht vorhanden, weil kein Standard den Einsatz solcher Verfahren forderte.

DARTT entstand als Werkzeug für Vorab-Entwicklertests. Mit der Zeit wurde das Werkzeug immer wieder um neue und aktuelle Konzepte erweitert. Herausgekommen ist ein integriertes Testwerkzeug, das inzwischen auch für Komponententests in der Europäischen Raumfahrt eingesetzt wird[GGB+05][Ger05a].

Inzwischen existiert auch ein entsprechendes Werkzeug für die Programmiersprache C namens DCRTT. Aus einem Prototypenwerkzeug namens SMARTG für eine Untermenge von Java, das im Rahmen der Diplomarbeit[Ger05b] des Autors entstanden ist, ergaben sich zudem weitere Ansätze für die selektive Reduktion der Testdatenmengen.

Im Folgenden soll über die Herausforderungen und Konzepte berichtet werden, die bei dieser Entwicklung entstanden sind.

2 Ziele

In der praktischen Anwendung wird das eigentliche Testziel, Fehler zu finden, noch durch ökonomische und organisatorische Anforderungen an den Prozess ergänzt. Zudem wird Test im Rahmen der Entwicklung von softwarebasierten Systemen oft nicht als konstruktive Maßnahme wahrgenommen, sondern als destruktive und eigentlich unnötige Pflicht, die keinen Beitrag zur Produkterstellung liefert und eventuell sogar das bereits erreichte zunichte macht. Dabei ist es gerade die analytische Qualitätssicherung, zu der der Softwaretest gehört, die sicherstellt, dass das erstellte Produkt auch dem geforderten entspricht. Sie macht also im Zweifelsfall das erreichte nicht zunichte, sondern stellt fest, ob überhaupt etwas erreicht wurde.

Aufgrund dieser Anforderungen muss der Testaufwand so gering wie möglich gehalten werden. Dieser macht sich in allen drei Phasen des Softwaretests bemerkbar und ist im Allgemeinen von der Anzahl der ausgeführten bzw. auszuwertenden Testfälle abhängig.

Zusätzlich müssen für das jeweilige Anwendungsgebiet geltende Mindeststandards erfüllt werden. Diese sind meist in Form von üblichen Abdeckungskriterien wie etwa Anweisungsüberdeckung oder MC/DC angegeben.

3 Strategien

Eine sehr einfache Möglichkeit, den Aufwand bei der Auswahl von Testeingaben zu reduzieren, ist der Einsatz von Zufallsgeneratoren. So können ohne großen Implementierungs- und Rechenaufwand große Testeingabemengen erzeugt werden. Allerdings werden hierbei Spezialfälle nur selten getroffen werden, die zwar für den Tester sehr interessant sind, aber nur einem kleinen Anteil am Eingaberaum entsprechen.

Deshalb analysiert DCRTT außerdem die im Programm vorkommenden Bedingungen,

und setzt die darin verwendeten Konstanten zusätzlich als Werte für Parameter ein. Dadurch kann die Abdeckung solcher Spezialfälle erhöht werden, auch wenn das Verfahren natürlich keine abschließende Lösung ist.

Zusätzlich werden Testeingaben aus den Schnittpunkten eines Gitters (Lattice) ausgewählt, das über die Eingabemenge gelegt wird. Hierbei zeigt sich insbesondere, dass die Testeingaben vom Rand des Eingaberaums besonders oft zum Auftreten von Ausnahmezuständen (Exceptions) im Prüfling führen und damit Fehler bei der Behandlung dieser Grenzfälle aufdecken[GGB+05].

Das Ergebnis ist eine große Menge an Testeingaben. So werden für die 140 Funktionen, die als Benchmark für das Werkzeug verwendet werden, insgesamt ca. 450.000 Testeingabetupel nach den oben angegebenen Methoden erzeugt. Hiermit sind über 85 % Blockabdeckung und über 95 % Entscheidungsüberdeckung (MC/DC) zu erreichen. Beim Einsatz eines Gitters werden über 90 % Blockabdeckung bzw. über 95 % Entscheidungsüberdeckung erreicht.

Vollständige Abdeckung kann bei den Benchmark-Funktionen nicht erreicht werden, da einige Zweige nicht erreichbar sind (dead code). In realen Projekten kann eine unvollständige Abdeckung ebenfalls auf Fehler im Code hinweisen.

Für den gesamten Testprozess auf Basis der Benchmarkfunktionen von der Vorbereitung über die Testdatenerzeugung bis zur Vorauswertung und Aufbereitung der Informationen für den Nutzer benötigt DCRTT auf einem handelsüblichen Pentium III mit 350 MHz 30 Minuten.

Die Ergebnisse einer solch großen Menge von Testeingaben ist natürlich nur schwerlich mit annehmbarem Aufwand manuell zu analysieren. Für die Gesamtmenge der Testeingaben wird deshalb nur eine statistische Analyse durchgeführt.

Um eine manuelle Analyse zu ermöglichen, wird aus der gesamten erzeugten Menge von Testeingaben nach bestimmten strategischen Gesichtspunkten eine Untermenge ausgewählt. Im Gegensatz zu den meisten Reduktionsverfahren geschieht dies nach der automatischen Einspeisung der Testeingaben und nicht bereits bei deren Generierung.

Nach der Ausführung ist etwa bekannt, ob eine Exception oder eine Zeitüberschreitung (etwa auf Grund eines Dead oder Life Lock) aufgetreten ist oder ob vorgegebene Maximalwerte für bestimmte Variablen überschritten wurden. Damit liegen nach der Ausführung genauere Informationen für die Selektion interessanter Testeingaben zur manuellen Auswertung vor als davor.

Die Auswahl erfolgt außerdem so, dass die Testdatenuntermenge auch die Abdeckungsanforderungen erfüllt. Dies ist aus organisatorischen Gesichtspunkten notwendig, da Auftraggeber oder relevante Industriestandards oftmals Vorgaben zur Testdurchführung machen, die ein Abdeckungskriterum bezüglich der einzeln ausgewerteten Testfälle beinhalten.

Für die Benchmarkfunktionen liefert die Selektion reduzierte Testdatenmengen mit ca. 690 Testeingabetupeln beim Einsatz eines Gitters über der Eingabemenge und ca. 620 Testeingabetupeln beim Einsatz von Zufallstestdatenerzeugung mit zusätzlichen Heuristiken. Der Unterschied hängt mit der Abhängigkeit des Reduktionsalgorithmus von der Reihenfolge

der Testdatentupel zusammen. Da sich die Reihenfolge bei gitter- und zufallsbasierter Testdatenerzeugung erheblich unterscheiden und Testeingabedaten auch mehrere Kategorien (z.B. unterschiedliche Anweisungen) abdecken können, können bei zufälliger Erzeugung oftmals kleinere Testfallmengen erreicht werden.

Die Anzahl der Elemente in der reduzierten Testdatenmenge unterscheidet sich auch je nach dem, welche Plattform eingesetzt wird, da unterschiedliche Compiler und Betriebssysteme unterschiedliche Exceptions erkennen, die dann zu einer unterschiedlichen Bewertung bestimmter Testeingaben führen.

4 Mechanismen

Durch die Ziele und Strategien ist der Anforderungsrahmen gesteckt. Sie beschreiben das "was" des Werkzeugs. Die tatsächliche Herausforderung bei der Umsetzung in die Praxis steckt jedoch im Detail. Dies sind die Mechanismen, die das "wie" des Werkzeugs beschreiben.

Eine der großten Hürden für die Umsetzung eines Werkzeugs für den automatischen oder teil-automatischen Softwaretest ist die Analyse des Programmcodes. Hier muss mit Sprachen wie C, Ada oder gar C++ gearbeitet werden, wie sie in der industriellen Praxis zum Einsatz kommen. Einschränkungen der zulässigen Programmkonstrukte sind nur in äußerst geringem Maße möglich.

Die Analyse ist gleich für mehrere Funktionen des Systems grundsätzliche Vorraussetzung. So werden ihre Ergebnisse benötigt, um die Schnittstellen des Prüflings festzustellen, ebenso wie für die Erstellung der Stimulationsumgebung zur automatischen Testausführung und die Aufzeichnung der Ergebnisse. Auch die Instrumentierung zur Messung von Abdeckung oder tatsächlichen Wertebereichen von Variablen und ähnlichen Aufzeichnungsfunktionen hängt von der Analyse ab.

Bei der Schnittstellenanalyse müssen sämtliche relevanten Typdeklarationen bekannt sein, um überhaupt Testdaten generieren zu können. In allen in der Praxis relevanten Programmiersprachen lassen sich die nutzerdefinierten Typen auf primitive Basistypen wie `int` in C zurückführen. Für die primitiven Typen können somit einfach Zufallsgeneratoren definiert werden. Für die zusammengesetzten Typen kann Code für entsprechende Funktionen erzeugt werden, der die einzelnen Felder der Strukturen belegt. Ähnlich werden Felder (Arrays) behandelt, wobei hier zunächst die Länge des Feldes zufällig bestimmt werden muss.

Besondere Probleme bereiten hierbei insbesondere die Zeigertypen von C und C++. Hier ist oftmals nicht klar, ob sich nicht eigentlich ein entsprechendes Feld hinter der Deklaration verbergen soll, was Auswirkungen darauf hat, wie Eingaben für den jeweiligen Parameter zu erzeugen sind. Auch ist nur schwer zu unterscheiden, ob der Parameter tatsächlich ein Eingabeparameter ist oder mit Hilfe des Zeigers als Ausgabeparameter verwendet wird.

Es ist außerdem nicht unüblich, dass zu jedem Feldparameter ein weiterer Parameter exis-

tiert, in dem die Länge des Feldes erwartet wird. Auch diese Korrespondenz kann aus der einfachen Analyse des Funktionskopfes in C oder C++ nicht erkannt werden.

Prinzipiell muss natürlich die durch den Funktionskopf gebotene Schnittstelle robust sein und auch auf Robustheit getestet werden. Somit müssen die Funktionen mit allen laut Funktionskopf zulässigen Eingaben korrekt umgehen können. Leider ist es gerade für Felder und Zeiger in C praktisch nicht möglich, innerhalb der Funktion eine Prüfung durchzuführen, so dass die aufgerufene Funktion die fehlerhafte Eingabe gar nicht als solche erkennen kann.

Dies führt aber auch und gerade beim Test zu Problemen. Die Korrespondenzbedingung zwischen Feldlänge und Wert des Längenparameters erfordert, dass zwei nacheinander zufällig ausgewählte Ganzzahlen — die Länge des Feldes und der Wert für den Längenparameter — den gleichen Wert erhalten. Damit werden jene Testeingaben unverhältnismäßig überbetont, die nicht der normalen und angedachten Verwendung der Funktion entsprechen.

Für die Erkennung von Ein- und Ausgabeparametern wurden die zusätzlichen Literale `_IN_`, `_OUT_` und `_INOUT_` eingeführt. Diese können einer Parameterdeklaration vorangestellt werden. Somit kann etwa bei Lattice-basierter Erzeugung der tatsächliche Eingaberaum mit derselben Zahl von Testeingabetupeln besser abgedeckt werden, wenn bestimmte Parameter als ausschließliche Ausgabeparameter erkannt werden. Ähnliche zusätzliche Informationen können für die Darstellung der Korrespondenz von Feldparameter und Feldlängenparameter verwendet werden.

Für die Programmiersprache Ada existieren Probleme dieser Art jedoch nicht. Hier muss die Bezugsrichtung der Parameter bereits aufgrund der syntaktischen Vorschriften der Sprache angegeben werden. Ebenso werden Felder explizit und mit vorgegebenen Grenzen deklariert. Es existiert keine Zeigerarithmetik wie in C oder C++, so dass bei Referenztypen genau bekannt ist, ob ein Feld oder nur ein einzelner Eintrag referenziert wird.

Für die Messung von Abdeckung und anderer Informationen aus dem internen Ablauf des Prüflings muss dieser instrumentiert werden. Dafür werden zusätzliche Anweisungen an relevanten Stellen — etwa am Beginn eines jeden Anweisungsblocks — eingefügt. Durch diese Anweisung wird, wenn sie ausgeführt wird, z.B. die Information aufgezeichnet, dass die instrumentierte Stelle passiert wurde, woraus sich Informationen über die Abdeckung ableiten lassen.

In Bereichen, in denen eine Instrumentierung nicht zulässig ist, etwa weil Vorschriften besagen, dass das System wie getestet auch in Dienst gestellt werden muss, müssen andere hardwareunterstützte Verfahren eingesetzt werden. So können Hardwareeinbauten den Adressbus des Zielsystems abhören und damit feststellen, auf welche Code- oder Datenelemente zugegriffen wird.

Dies schließt jedoch nicht aus, dass zuvor dennoch mit instrumentierter Software getestet wird. Hier geht es insbesondere darum, die verfügbaren Methoden soweit möglich auszunutzen, um zu Ergebnissen und Informationen zu kommen. So können etwa die Information, die aus dem Test des instrumentierten Systems gewonnen wird, durch Analyse auf das nicht instrumentierte System übertragen und somit überprüft werden, ob tatsächlich ein Problem im System vorliegt.

Die Reduktion der Testdatenmenge erfolgt, indem die verschiedenen interessanten oder zwingend abzudeckenden Elemente — etwa Exceptions (interessant) oder zusammenhängende Anweisungsgruppen (zwingend abzudecken) — identifiziert werden. Jedem solchen Element wird eine Zahl an zu sammelnden Testdatentupeln zugeordnet.

Bei jedem Testdatentupel wird nun nach der Einspeisung in den Prüfling identifiziert, welche der Elemente durch dieses Tupel abgedeckt werden. Ein Tupel wird nur dann in die reduzierte Menge aufgenommen, wenn es mindestens ein zuvor noch nicht ausreichend abgedecktes Element abdeckt.

5 Methoden

In den vorigen Kapiteln wurden die Details der Werkzeugimplementierung betrachtet. Unter den Methoden sind im Gegensatz dazu die Details der Werkzeuganwendung zu verstehen.

Das kombinierte Verfahren aus zufälliger Erzeugung großer Testeingabemengen, automatischer Testausführung und automatisch unterstützter Auswertung mit Selektion einer reduzierten Testdatenmenge erfordert eine grundlegend andere Vorgehensweise des ausführenden Qualitätssicherers als der manuelle Test.

Es darf nicht unerwähnt bleiben, dass gerade dieser Unterschied zu einem Akzeptanzproblem führen kann. Der Zwang zur konzeptionellen Umstellung verbunden mit der Angst, die eigene Arbeit werde durch den Computer ersetzt, erzeugen ein ablehnendes Klima. Diese psychologischen Probleme können weder durch theoretische Grundlagenforschung noch durch ein noch so gut implementiertes Werkzeug gelöst werden. Hier ist psychologisches Geschick des Projekt- und Firmenmanagements gefragt.

Von einer arbeitsplatzzerstörenden Rationalisierung in der Qualitätssicherung kann jedoch keine Rede sein. Vielmehr übernimmt der Computer mehr reine Fleißarbeit, während dem Bediener eine deutlich kreativere Aufgabe zukommt, nämlich, den Testprozess zu steuern. Der Qualitätssicherer ist damit in der Lage, seine eigentliche Aufgabe effizienter auszuführen und damit eventuell sogar seinen Arbeitsplatz zu sichern.

Die Steuerung kann auf mehreren Ebenen erfolgen. Sowohl die Erzeugung von Testeingabedaten als auch die Auswertung und die Selektion der reduzierten Testdatenmenge können gelenkt werden. Die Ausbildung muss diese Methoden vermitteln, so dass Qualitätssicherer in der Lage sind, für den einzelnen Prüfling jeweils entsprechende Steuerungs- und Auswertungskonzepte zu entwickeln und beim Test mit großen Testeingabemengen und nicht mehr nur mit einzelnen Testfällen zu hantieren.

5.1 Steuerung der Generierung

Manchmal ist die Ausführung bestimmter Sequenzen von Aufrufen unterschiedlicher kooperierender Funktionen beim Test gewünscht. So ist es beim Test einer Stack-Implemen-

tierung zwar sicherlich interessant, ob die Implementierung korrekt mit einer `pop`-Operation auf einem leeren Stack umgeht. Für die weiteren Tests dürfte es aber eher von Interesse sein, wie die Funktion sich auf einem gefüllten Stack verhält.

Um solche Sequenzen zu erzeugen und zu testen, kann der Tester eine Sequenzierungsfunktion vorgeben, für die dann zusätzlich Eingabedaten erzeugt werden. Ein Beispiel für den genannten Stack könnte etwa so aussehen:

```
struct seq_entry { action_t action; int datum; };
stack_t* test_stack_sequence(int first_element,
                             struct seq_entry seq[]) {
  struct seq_entry* entry;
  stack_t* stack=new_stack();

  push(stack,first_element);
  for (entry=sequence;entry->action!=END;entry++) {
    if (entry->action==PUSH) push(stack,entry->datum);
    else if (entry->action==POP) pop(stack);
  }

  return stack;
}
```

Auf ähnliche Weise können auch bestimmte Untermengen der Eingabemenge gezielt angesteuert werden. Hierfür ist eine Parameterisierung dieser Untermenge aufzustellen. Bei der Berechnung des größten gemeinsamen Teilers zweier Zahlen ist etwa der Spezialfall interessant, bei dem die beiden Zahlen gleich sind. Hierfür könnte etwa die folgende Funktion vorgeschaltet werden:

```
int ggt_pair_test(int a) {
  return ggt(a,a);
}
```

Bei der Testdatenerzeugung wird für diese Funktion nur ein einzelner Parameterwert erzeugt, der dann durch die Funktion selbst dupliziert wird. Der eigentliche Prüfling wird so also speziell mit Eingaben aus der vorgegebenen Untermenge ausgeführt.

Das vorgestellte Werkzeug kann auch allein unter Angabe des unveränderten Quellcodes angewendet werden und liefert zielführende Ergebnisse[GGB+05][Ger05a].

Um den Prozess gezielt steuern zu können, ist hier ggf. Vorarbeit notwendig. Sie unterscheidet sich grundsätzlich von der Vorarbeit, die beim manuellen Test erforderlich ist, insbesondere dadurch, dass nun prinzipiell beliebig große Schaaren von Testeingaben spezifiziert werden, statt nur einzelne Testeingaben zu beschreiben. Die tatsächliche Testdatenmenge kann also jederzeit erweitert oder gar vollständig ausgetauscht werden, ohne dass die Vorarbeit neu auszuführen wäre.

5.2 Auswertungsmethoden

Mit demselben Prinzip können auch Plausibilitäts- oder gar Korrektheitsprüfungen eingesetzt werden. Dafür wird um den Prüfling erneut eine zusätzliche Funktion eingeführt, die die gleiche Parameterdeklaration erhält wie der Prüfling selbst. Diese Funktion ruft den Prüfling auf und prüft das Ergebnis. Ist das Ergebnis plausibel, wird es einfach von der umgebenden Funktion an den Testprozess zurückgeliefert. Ist es nicht plausibel oder gar nicht korrekt, so kann die Funktion eine speziell dafür vorgesehene Exception auslösen und damit für den Testprozess das jeweilige Testdatum und sein Ergebnis als besonders interessant markieren. Damit wird dieses Testdatum mit erhöhter Wahrscheinlichkeit, oder — je nach Implementierung — mit Sicherheit in die reduzierte Testdatenmenge aufgenommen und zudem für den Auswerter noch hervorgehoben.

In einer Erweiterung des Prinzips sind gar probabilistische Prüffunktionen denkbar, etwa, wenn stochastische Eigenschaften der zu testenden Funktion bekannt[MG04] sind oder die Funktion höchst kompliziert ist, aber das Ergebnis z.B. mit Monte-Carlo-Methoden näherungsweise überprüft werden kann. Die angegebene Wahrscheinlichkeit für einen Fehler wird dann bei der Selektion der reduzierten Testdatenmenge angewendet, um zu entscheiden, ob die jeweilige Testeingabe in die reduzierte Testdatenmenge aufgenommen wird. Weitere Methoden finden sich auch etwa in der Ausnutzung von Symmetrien zu testender Systeme[Got03] zur Überprüfung der Ergebnisse.

Es ist festzustellen, dass beim Testen in großen Maßstäben, wie es mit dem vorgestellten Werkzeug betrieben wird, nun auch neue Herausforderung bei der Auswertung der Testergebnisse aufkommen. Schließlich werden beim Test hier große Datenmengen produziert, die es zu ordnen und zu interpretieren gilt.

Bisher führt lediglich das gute Auge und der gute Sinn für bessere Darstellungsformen des ausführenden Qualitätssicherers die Auswertung. Klar definierte und theoretisch untermauerte Auswertungsformen fehlen größtenteils, auch wenn bereits einige vielversprechende Ansätze untersucht wurden.

Gesucht sind nun weitere Verfahren, mit denen Auswertungen großer Mengen an Testausführungsergebnissen auf Hinweise auf Fehler untersucht werden können.

5.3 Aufwand und Aufwandsreduktion

Für umfangreiche Systeme mit tausenden von Funktionen[GGB+05] ist es natürlich nicht machbar, solche Assistenzfunktionen für jede einzelne zu testende Funktion bereitzustellen. Insofern liegt auch hier eine weitere Herausforderung, Verfahren zu erkunden, mit denen die Ableitung solcher Funktionen automatisiert werden kann.

Ein Ansatz basiert darauf, bestimmte Anforderungen an die zu testenden Abläufe im Prüfling zu stellen. Dabei wird für jeden Prüfling ein neues Abdeckungskriterium definiert. Dieses Abdeckungskriterium wird jedoch nicht rein manuell erstellt, sondern basiert auf einem Regelsystem, das in Abhängigkeit von der Struktur des Prüflings die zu tes-

tenden Pfadmengen konstruiert. Vorgabe ist dabei, dass zu jeder konstruierten Pfadmenge eine gegebene Anzahl von Testeingaben zu finden sind, bei deren Einspeisung ein Pfad aus der jeweiligen Pfadmenge durchlaufen wird[Ger05b].

Die Regelstruktur kann dabei angepasst und erweitert werden, um auf spezifische Anforderungen des einzelnen Prüflings zu reagieren oder hinzugewonnenes Know-How über interessante Teilabläufe im Rahmen sich ähnelnder Systeme eines Anwendungsgebiets zu sichern.

6 Schlussbemerkungen

Die Ergebnisse der theoretischen Forschung zum Thema Testautomation finden sehr wohl Eingang in die praktische Arbeit und viele Forschungsbeiträge führen auch in der Praxis zu sehr guten und interessanten Ergebnissen.

Die Herausforderung bei der Umsetzung der theoretischen Konzepte in die Praxis liegt jedoch oft darin, dass einige Vereinfachungen, die in der Forschung angenommen werden, in der Praxis nicht zulässig sind und überwunden werden müssen. Allein die Realität der aktuellen Landschaft der Programmiersprachen verlangt dem Implementierer großen Aufwand ab, bevor überhaupt an eine korrekte Umsetzung der Forschungsergebnisse gedacht werden kann.

So schränkt etwa Gotlieb[GBR98] die zulässigen Programmkonstrukte in seiner ansonsten richtungsweisenden Arbeit zur automatischen Testdatenerzeugung auf Basis von Constraintlöserverfahren auf `while`- und `if-then-else`-Konstrukte ein. Dies ist zwar keine tatsächliche Beschränkung der Allgemeinheit, da sich andere Konstrukte hierauf abbilden lassen. Es ist aber eine Beschränkung der Einsetzbarkeit, die bei der Implementierung zunächst zu überwinden ist. Eine Verallgemeinerung genau jenes Konzeptes befindet sich im übrigen gerade im Entwicklungsstadium.

Softwaretest ist kein absolutes Qualitätssicherungsinstrument. Die minimale Testfallmenge mit der alle Fehler gefunden werden können, mag zwar existieren, aber es ist nicht möglich, sie direkt abzuleiten. Somit wird das Vorgehen beim Softwaretest immer auf Strategien ohne Erfolgsgarantie basieren. Ziel muss es daher sein, das Machbare möglich zu machen und ggf. auch mit unvollständigen Verfahren Informationen zu sammeln. Eine Kritik allein auf Basis dessen, was ein Verfahren nicht zu leisten in der Lage ist, ist daher kontraproduktiv.

Gesucht sind jetzt Mechanismen und Methoden, die den Anwender bei der Testvorbereitung und -auswertung unterstützen und leiten, etwa nach den Gesichtspunkten typischer Fehler und Fehlerquellen. Dabei verlässt die Forschung den Bereich der algorithmenorientierten Informatik und betritt auch das Gebiet der Ergonomie und der Projektauswertung. Insbesondere für letzteres wird ein langer Atem benötigt, denn trotz des jungen Alters ist die Informatik bereits stark ökonomisiert und Daten für Projektstudien werden — wenn überhaupt — nur widerwillig bereitgestellt. Aus diesem Grund können im Fall des vorgestellten Werkzeuges auch leider nur Ergebnisse aus eigenen Projekten präsentiert werden.

Aus demselben Grund ist es auch so schwer, neue und aktuelle Methoden aus der Forschung in die Praxis einzuführen. Für Firmen ist es im Konkurrenzkampf nicht gewünscht zuzugeben, dass Probleme oder gar bessere Vorgehensweisen als die aktuell verwendeten existieren. Auch findet man insbesondere bei den etablierten Herstellern entsprechender Werkzeuge oftmals eine sehr konservative Haltung im Bezug auf neue Verfahren vor. Diese möchten ihre Rendite schützen und haben somit kein Interesse daran, ihr gut laufendes Geschäft auf neue Konzepte umzustellen.

Abhilfe könnte die Förderung eines Wettbewerbs zwischen den Werkzeugen schaffen, der nur auf vergleichbaren Effizienzkriterien basieren sollte, also etwa den zur Anwendung benötigten Ressourcen und der Rate der gefundenen Fehler.

Die Dissemination von Forschungsergebnissen in die Praxis kann deshalb nur über den marketinggerechten Nachweis der Praxistauglichkeit erfolgen. Die Anforderungen hierfür sind klar: Tests für große und kleine Systeme bei gleichzeitiger Aufwandsreduktion in allen Fällen und ohne Einschränkungen der Freiheiten bei der Anwendung der existierenden Entwicklungswerkzeuge und -sprachen.

7 Danksagung

Ralf Gerlich möchte sich bei der Universität Ulm für die Förderung seiner Forschungsarbeit zum Thema durch ein Promotionsstipendium bedanken.

Literatur

[GBR98] Arnaud Gotlieb, Bernard Botella und Michel Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, 1998.

[Ger05a] Rainer Gerlich. DARTT Test Results AISVV-FAS. Bericht, BSSE, 2005. Automated ISVV, ESTEC contract no. 18056/04/NL/JA.

[Ger05b] Ralf Gerlich. Size-Optimising Automatic Random Testcase Generation for Non-Formal Conditions. Diplomarbeit, Universität Ulm, 2005.

[GF93] Rainer Gerlich und G. Fercher. A Random-Testing Environment for Ada Programs. In *Eurospace Symposium 'Ada in Aerospace'*, 1993.

[GGB+05] Rainer Gerlich, Ralf Gerlich, Thomas Boll, Klaus Ludwig, Philippe Chevalley und Neil Langmead. Software Diversity by Automation. In *Proceedings of DASIA 2005 - DAta Systems In Aerospace*, 2005.

[Got03] Arnaud Gotlieb. Exploiting Symmetries to Test Programs. In *ISSRE'03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, 2003.

[MG04] Johannes Mayer und Ralph Guderlei. Test Oracles using Statistical Methods. In *Testing of Component-Based Systems and Software Quality*, 2004.