Dr. Hauke Reddmann,
Hausarbeit Informatik-Seminar
„Softwareentwicklung in der Wissenschaft“
(C. Hovy), Universität Hamburg SoSe 16

„If it walks like a duck“

Softwaretests für untestbare Software

Inhaltsverzeichnis

Motto: Modus barbara
„Die Geschichte des Testens ist die Geschichte der Testautomatisierung.“ [Sneed12]
„Die Geschichte der Testautomatisierung ist eine Geschichte voller Missverständnisse.“ [Seidl12]
„Die Geschichte des explorativen Testens ist eine Geschichte voller Missverständnisse.“ [Hendrickson14]

1. Einleitung

1.1. **Thematik, Zielsetzung und methodischer Aufbau**

1.1.1. Thematik

Nicht oder nicht wie geplant funktionierende Software in der Wissenschaft hat zwar wohl nicht das gleiche Potential für katastrophales Versagen wie anderswo, ist aber natürlich genauso unerwünscht. Daher sollte auch wissenschaftliche Software ausgiebig getestet werden. Allerdings ergeben sich aus dem Kontext „Wissenschaft“ meist einige Besonderheiten, speziell die Möglichkeit, daß mit (industrie)üblichen „Standard“-Testmethoden gar keine sinnvollen Tests durchgeführt werden können.

1.1.2. Zielsetzung

Ziel soll es sein, die Theorie des Softwaretestens darzulegen und zu erläutern, welche speziellen Voraussetzungen bei wissenschaftlicher Software (meist) erfüllt sind und was dies für das Testen bedeutet, sowie einige Testmethoden näher vorzustellen.

1.1.3. Methodischer Aufbau

Die Einleitung führt kurz in die Problematik ein. Danach folgt der theoretische Teil, der die Begrifflichkeiten und verschiedenen Kategorien von Softwaretests erläutert und auf die Unterschiede eingeht. Es folgen einige ausgesuchte Praxisbeispiele. Die Arbeit hält sich lose an die 6W der Journalistik [Wiki6W].

1.2. **Softwaretests**

WARUM?
Gar nicht oder nicht wie geplant funktionierende Software ist ein ständiges Ärgernis für Programmierer, Anwender und Industrie. Das klassische Beispiel für ein sehr teures und lautstarkes Versagen wegen eines eigentlich trivialen Anfängerfehlers ist das Ariane-Desaster [WikiAriane], was in keiner Arbeit über Softwaretesten fehlen darf. Leiser, aber fast täglich machen sich die unvermeidlichen Fehler in jedem beliebigen [MS] OS mit sehr vielen Codezeilen bemerkbar. Es folgen einige grobe Abschätzungen (ohne Gewähr) aus [Sneed12] und dort angegebenen Literaturstellen:

- Roher Code hat eine Fehlerdichte (Fehler pro Codezeile) von 1%.
- Gut getesteter Code hat eine Fehlerdichte von 0.3%.
- Diese Werte haben sich trotz aller neuer Softwareparadigmen, Verbesserungen und Erfindungen von Tools zur Fehlersuche nicht geändert. Allenfalls bekommt man jetzt mehr Funktionalität pro verbleibenden Fehlern.
- Der Schaden durch Softwarefehler in der Automobil- und Flugzeugbranche der USA beträgt 16% des Softwarewertes.
- Um alle Fehler in deutscher Software zu beheben, wäre 9% des Softwarewertes nötig.

Über das „Warum“ von Softwaretests gibt es also inzwischen keine Diskussion mehr. Allenfalls „Wegwerfsoftware“ (oder höflicher: Einmalsoftware – in der Wissenschaft kann es ja durchaus vorkommen, daß man nur an einer Lösung interessiert ist, und hat das Programm diese geliefert, wird es nie wieder ausgeführt), die niemals wiederverwendet oder editiert wird, könnte ungetestet bleiben.
Klassische Fehlerquellen in Software sind z.B.: Noch ungetesteter Code, andere Reihenfolge der Exekution von Statements während des Betriebs sowie ungetestete Kombinationen von Input und Umgebungsvariablen wie Betriebssystem, Bibliotheken, Peripherie und Netzwerk [Whittaker00] (der Autor gibt auch eine Reihe von „Standard“-Programmierfehlern sowie eine Checkliste zu ihrer Vermeidung an). Laut einer alten Programmierweisheit gilt als funktionierende Software solche, bei der man noch nicht herausgefunden hat, welcher Input sie zum Abstürzen bringt [LB]. (Noch schlimmer: Spekulativen Überlegungen zufolge impliziert Gödels Theorem [Gödel31], daß für hinreichend komplexe Programme ein solcher Input existieren muss.)
Natürlich wird auch in der Wissenschaft Software benutzt (und das weit mehr als der Durchschnittsbürger vermutet – der Autor dieser Arbeit ist selbst das beste Beispiel: wenn er sagt, er ist Chemiker, muss er erst einmal lange erklären, warum er 100% seiner Zeit am Computer verbringt, da dies natürlich nicht der landläufigen Vorstellung von einem Chemiker entspricht). In der Wissenschaft neigen die Fehler dazu, subtil und hinterhältig zu sein.
Softwaretests sollen die Fehler gleich an der Quelle beseitigen. Doch in der Wissenschaft wissen wir nicht von vornherein, was „hinten rauskommt“ (sonst wäre die Forschung ja gar nicht erst nötig). Dies kann sogar bis zum paradoxen Effekt führen, daß Software völlig korrekt funktioniert, den Ergebnissen aber nicht geglaubt wird [WikiOzon].
Also: Wie testet man Software? Was macht Software „untestbar“? Wie testet man untestbare (wissenschaftliche oder auch andere) Software trotzdem? Im Hauptteil dieser Arbeit werden hierzu Möglichkeiten vorgestellt.

2. Theorie

2.1. **Fehler**

WAS?
[Spillner12] definiert einen Fehler als die Nichterfüllung einer festgelegten Anforderung bzw. eine Abweichung zwischen dem Istverhalten (während Tests oder Betrieb) und dem Sollverhalten (Spezifikation oder Anforderungen). Wurde also gar kein Sollverhalten vorgegeben, gilt eine klassische Entwicklerweisheit: „It’s no bug, it’s a feature!“ [LB].
Ein Programm kann auch fehlerhaft sein, ohne daß sich der Fehler zeigt (zwei kompensierende Fehler, Vorbedingung für Fehler noch nie eingetreten etc. – die englische Sprache unterscheidet hier etwas genauer zwischen „fault“, der Fehlerursache, „error“, dem Fehlerzustand, und „failure“, der Fehlerwirkung [Mili15]). Somit gilt der Grundsatz [Spillner12]:

> Fehler lassen sich nur nachweisen, aber nicht ausschließen.

Der „philosophische“ Umgang mit Fehlern variiert stark: Einerseits wird versucht, Software zu erstellen, die noch vorhandene Fehler tolerieren kann [Cristian85], andererseits existieren Tools, die künstlich Fehler zu einem existierenden Programm hinzufügen, um den Programmierer aufmerksam zu halten und die Korrektheit von bereits geschriebenen Tests (s. Punkt 2.3) zu testen [WikiBug].

2.2. **Softwarequalität**

WAS?

Softwarequalität umfasst weit mehr als rein technische Fehlerfreiheit [Seidl12]. Gemäß ISO9126 [ISO9216] bzw. dem Nachfolger ISO29119 [Daigl16] zählt man folgende Punkte auf:

* Externe Qualitäten
    - Funktionalität: Angemessenheit, Richtigkeit, Interoperabilität, Sicherheit, Ordnungsmäßigkeit
    - Zuverlässigkeit: Reife, Fehlertoleranz, Wiederherstellbarkeit
    - Benutzbarkeit: Verständlichkeit, Erlernbarkeit, Bedienbarkeit, Attraktivität
    - Effizienz: Zeitverhalten, Verbrauchsverhalten
* Interne Qualitäten
    - Änderbarkeit: Analysierbarkeit, Modifizierbarkeit, Stabilität, Testbarkeit
    - Übertragbarkeit: Anpassbarkeit, Installierbarkeit, Koexistenz, Austauschbarkeit
* Konformität: Übereinstimmung mit den Spezifikationen (zählt zu allen sechs vorigen Hauptpunkten)

2.3. **Testen**

WAS?

[Spillner12] definiert Testen von Software als Ausführung (im Allgemeinen stichprobenartig; es gibt im Übrigen auch statische Tests ohne Ausführung, s.u.) eines Testobjektes zu seiner Überprüfung unter festgelegten Randbedingungen. Ob das Testobjekt die geforderten Eigenschaften erfüllt, wird hierbei durch einen Vergleich zwischen Soll- und Istverhalten bestimmt.

[Spillner12] gibt die folgenden sieben Grundsätze zum Testen an:

- Testen zeigt die Anwesenheit von Fehlern.
- Vollständiges Testen ist nicht möglich.
- Tests sollen so früh (im Softwarezyklus) wie möglich beginnen.
- Fehler treten clusterweise auf.
- Fehler neigen dazu, gegen Testen resistent zu werden.
- Testen ist abhängig vom Gesamtsystem.
- „Keine Fehler“ bedeutet nicht „brauchbares System“.

Testen findet Fehler(wirkungen), bestimmt die Qualität des Programms, erhöht das Vertrauen in das Programm und beugt auch Fehlerwirkungen vor [Spillner12]. [Myers04] schätzt, daß in Softwareprojekten die Hälfte der Zeit und mehr als die Hälfte der Kosten auf das Konto von gründlichem Testen gehen.

Nicht alle unter 2.2. genannten Punkte lassen sich gleich einfach testen, und ein weit verbreitetes Missverständnis ist, unter „Testen“ nur den Punkt „Richtigkeit“ (s.o.) auf der Stufe „Unit-Test“ (auch diese Arbeit konzentriert sich auf diese Aspekte) zu verstehen. Gegenüber dem älteren „Wasserfall-Modell“ stellt das neuere „V-Modell“ [V] aber heraus, daß Testen auf allen Stufen der Programmentwicklung (vom Entwurf bis zur Auslieferung) stattfindet.

Man darf Testen auch nicht mit Debuggen verwechseln: Um einen Fehler zu debuggen, muss man ihn erst einmal gefunden haben [Myers04].

WANN? WER? WO?
Basierend auf (u.a.) der Psychologie des Testens [Myers04] gibt es viele Empfehlungen zur Durchführung von Tests. Speziell seien genannt:

- Wann: Testfälle sollen schon *vor* dem eigentlichen Programmieren konzipiert werden.
- Wer: Der Test sollte nicht vom Programmierer selbst durchgeführt werden.
- Wo: Der Test sollte nicht einmal von der betreffenden Organisation durchgeführt werden.

Wiewohl es gute Gründe für jeder dieser Empfehlungen gibt (Testen soll Fehler aufdecken, und das Ego eines Programmierers bzw. im übertragenen Sinne einer Organisation sträubt sich instinktiv gegen das Finden von Fehlern in ihren eigenen Programmen), kann man mit gleicher Berechtigung diese Empfehlungen als absurd empfinden (Unit-Tests werden meist schon vom Entwickler erledigt, schließlich kennt der Programmierer seinen Code am besten). Bei wissenschaftlicher Software wird der letzte Punkt ganz besonders wichtig: wie will ein externer Tester Code testen, zu dessen Verständnis (lies: des Warums, nicht des Wie des Algorithmus) es eines Doktortitels bedürfte [Kelly10]?

WIE?
Es gibt eine verwirrende Vielfalt von Namen für Tests [Spillner12], in Abhängigkeit von deren:

- Ziel
- Methode
- Objekt
- Stufe
- Person
- Umfang

[Spillner12] folgend soll ein wahlloses Beispiel für jeden dieser Punkte gegeben werden:
Ziel: Lasttest: Der Test prüft, wie das Programm mit sehr großen Datenmengen umgeht.
Methode: Black-Box-Test: Die Testfälle werden ohne Kenntnis der internen Rechnungen eines Programms durchgeführt.
Objekt: GUI-Test: Die graphische Benutzerschnittstelle wird getestet.
Stufe: Unit-Test: Nur eine Programmeinheit wird separat getestet. (Die meisten Beispiele in dieser Arbeit fallen unter dieses Stichwort.)
Person: Anwendertest: Tests, die vom Anwender durchgeführt werden.
Umfang: Regressionstest: Wiederholung des gleichen Tests nach einem Update.
Des weiteren unterscheidet [Spillner12] noch zwischen struktur- und änderungsbezogenen Tests sowie zwischen funktionalen und nicht funktionalen Tests. Der letztere Unterschied lässt sich umschreiben als der zwischen dem Testen des „Was?“ (was leistet das System, und tut es das richtig und im Einklang mit den Anforderungen?) und das Testen des „Wie?“ (wie gut erfüllt das System die Anforderungen). Klarerweise sind letztere Anforderungen schwierig in Worte zu fassen. Daher lässt sich ein „Testorakel“ (s. Punkt 2.4) nur für erstere Tests sinnvoll definieren.
Testen ist auch eine Kunst (ohnehin können Menschen und Computer andere Typen von Fehlern am besten erkennen [Myers04]), und daher ist es nicht immer möglich, eine Automatisierung von Tests [Seidl12] (was Zeitersparnis und Qualitätserhöhung bedeutet) durchzuführen. Dies gilt natürlich ganz besonders im Bereich der wissenschaftlichen Software.
Das neuste Paradigma des „Agilen Testens“ [Baumgartner13] kann hier allenfalls am Rande erwähnt werden.

Im Rahmen dieser Arbeit soll eine weitere selbst entwickelte nützliche Klassifizierung kurz vorgestellt werden, eine „Mohs-Härteskala“ für Tests.

a) Sehr hart
Endgültige Wahrheit und Fehlerfreiheit ist selbst in der Mathematik unmöglich, wie die bahnbrechende Arbeit von Gödel [Gödel31] bewies: für jedes (hinreichend mächtige) Axiomensystem gibt es wahre Sätze, die nicht von diesem System bewiesen werden können. Daher können wir nur praktisch vorgehen: Was von Mathematikern peer-reviewt und für korrekt befunden wurde, soll als wahr gelten.
b) Hart
In diesem Sinne kann ein einfacher Algorithmus wie z.B. das Sortieren einer Liste als „mathematisch korrekt“ gelten, auch wenn er natürlich auch erst einmal korrekt implementiert werden muss (auch die Implementation selbst kann übrigens mathematisch als korrekt nachgewiesen werden [Mili15], natürlich wieder nur „jenseits vernünftiger Zweifel“). Eine solche Analyse fällt unter den Begriff „statischer Test“, im Gegensatz zum „dynamischen Test“ [Spillner12].
c) Mittel
Die meiste Software ist viel zu kompliziert, um durch eine algorithmische Analyse als korrekt nachgewiesen zu werden (ein Problem, was durch das Paradigma der objektorientierten Programmierung keinesfalls entschärft wird: das Chaos der interagierenden GOTOs wird lediglich ersetzt durch das Chaos der interagierenden Klassen [Baumgartner13]). Es müssen also Testfälle geschrieben werden (natürlich können diese nicht wirklich alle Möglichkeiten abdecken) und das Resultat mit der Spezifikation des Programms verglichen werden („Testorakel“, s. Punkt 2.4) [Spillner12]. Dies ist der „Normalfall“ in der Programmierung.
Ein weiteres Problem, was (nicht nur) beim wissenschaftlichen Rechnen auftauchen kann, ist die numerische Genauigkeit. Diese zeigt sich z.B. beim Abschneiden von unendlichen Serien, Runden von reellen Werten oder bei der Fehlerfortpflanzung [Chen02]. Ein korrekter Algorithmus kann immer noch numerisch instabil sein [Cox99]. Auch dies muss getestet werden.
d) Weich
Programme können aus verschiedenen Gründen „untestbar“ sein: die Antwort ist völlig unbekannt und das Programm wurde genau dafür geschrieben (ein sehr häufig vorkommender Fall in der Wissenschaft), oder der Output des Programms ist viel zu groß (man denke an Klimaforschung oder Hochenergiephysik), oder das Programmierer hat die Spezifikationen des Problems missverstanden [Davis81]. Wie also kann man testen, wenn nicht einmal ein Testorakel vorliegt? Hier kann man aber immer noch auf „paralleles“ oder „metamorphes“ Testen zurückgreifen, oder heuristische, statistische, Sampling- oder andere Methoden einsetzen, die in Kapitel 3 näher beleuchtet werden.
e) Sehr weich
Möglicherweise muss schon allein aus finanziellen und zeitlichen/Manpower-Gründen (Wissenschaftler sind chronisch unterfinanziert und kommen allein schon vor Bürokratie nicht mehr zum Forschen) das Testen fast komplett entfallen und es zählt wirklich nur noch, was „hinten rauskommt“. Beispiele aus eigener Erfahrung werden ebenfalls in Kapitel 3 vorgestellt.

2.4. **Testorakel**

WAS?

Nehmen wir also an, unser Programm (wissenschaftlich oder nicht) liefert das Ergebnis „239“. Jetzt wäre es gut zu wissen, ob „239“ auch das ist, was „eigentlich“ herauskommen sollte. Was kann uns diese Information liefern? Das „Testorakel“ (zu Details der Orakel-Terminologie s. [Richardson92]). Die folgende Aufzählung soll (an sehr vereinfachten Beispielen) nur die Prinzipien erläutern. In der Literatur herrscht eine ziemliche taxonomische Sprachverwirrung (vgl. auch [Bolton]) hinsichtlich Orakel, Pseudo-Orakel und Nicht-Orakel. Ist also z.B. das Orakel eine äußere Information (wie eine Rechnung per Hand) oder eine innere Subroutine (der Programmteil, der mit dieser Information vergleicht)? In dieser Arbeit soll streng nach der Herkunft des Begriffs definiert werden: Ein Orakel sagt *immer* die Wahrheit (ist also „korrekt“ und „vollständig“ gemäß der mathematischen Definition in [Staats12]), hier also Testfall bestanden/nicht bestanden. (Selbstverständlich schließt dies nicht aus, daß bei der Erstellung des Orakels selbst Fehler gemacht wurden, dann ist es aber kein Orakel, sondern ein Debakel – wie zum Beispiel beim erwähnten Ozonloch, wo die einschränkende Annahme „Zu niedrige Werte müssen ein Messfehler sein“ nur ein Vorurteil war.) Nach dieser Ansicht ist es egal, wo das Orakel im Gesamtsystem residiert und ob es uns direkt die Lösung liefert (aus einer Tabelle, einer Dokumentation, einer Spezifikation…) oder nur die Ja/Nein-Entscheidung (wie eine Runtime Assertion). Das Orakel darf wie sein antikes Gegenstück also „unverständlich“ sein. In der Wissenschaft kennen wir die Lösung meistens nicht, daher würde ein hypothetisches Orakel ggf. das ganze Programm überflüssig machen.

Ein Pseudo-Orakel gibt uns wenigstens nützliche Informationen über den Test, aber kein klares ja/nein. (Darunter fällt auch z.B., wenn es zwar ein „Nein, Test nicht bestanden“ liefern kann, aber kein „Ja“.) Ein Nicht-Orakel tut…nichts. (Fast, s.u.)

Es folgt eine knapp erläuterte Liste ohne Anspruch auf Vollständigkeit (vgl. [Hoffman98] für weitere Beispiele).

* Orakel

- Spezifikation/Dokumentation [Peters94]. Wir haben eine Methode, die laut Handbuch eine Addition durchführen muss. Wir schreiben einen Testfall mit dem Input „Plus(200, 39)“. Wir können also sofort sagen, daß mit „239“ der Test bestanden wurde. Dieses Orakel hat noch verschiedene Varianten wie „Gelöstes Beispiel“, „Simulation“, „Goldstandard“ etc. [Binder00] und ist umso effektiver, je formaler die Umsetzung von Dokumentation zu Orakel geschieht (wünschenswert ist auch noch, daß die Spezifikation nicht von der Implementation abhängt, und kein Spezialwissen zum Verständnis erfordert) [Peters94]. Eine spezifische Variante dieses Prinzips sind die aus Java bekannten „Runtime Assertions“, die das Ergebnis prüfen, aber nicht notwendigerweise vorgeben (und daher von anderen Autoren nur als Pseudo-Orakel gewertet werden) und somit auch bei „untestbaren“ Programmen (wissenschaftliche Programme sind im Vergleich bekanntermaßen notorisch schlecht dokumentiert und kommentiert [Clune11]) anwendbar sind.
- Menschliches Orakel [Ammann08]. Wie eben, aber wir stellen stattdessen in mühsamer Handarbeit fest, daß 200+39=239 ist, und haben damit einen Test als bestanden abgehakt. (Natürlich ist auch die Software des Gehirns nicht fehlerfrei…)
- Modellbasiert [Robinson99]. Wir bauen ein Modell des Programms und können mit einem statischen Test beweisen, daß die Methode „Obskur(200,39)“ nichts anderes tut, als die Werte zu addieren (was uns vorher nicht klar war).

* Pseudo-Orakel
    - Heuristik [Hoffman99]. Nehmen wir an, unsere Methode sei stattdessen „Sin(X)“. Ohne auch nur ins Detail zu gehen – der Test ist durchgefallen, denn der Sinus liegt immer zwischen -1 und 1. (Ob ein Test bestanden hat, können wir nicht sagen.)
    - Konsistenz [Hoffman98]. Lassen wir den Test ein paarmal mit leicht variablem Input durchlaufen. 239, 240, 241, 242…sieht gut aus, wir wissen zwar nicht, was da passiert, aber es ist in sich konsistent.
    - Statistik [Mayer04]. Lassen wir den Test noch ein paarmal mit leicht variablem Input durchlaufen. 239, 240, 239, 238, π…Es ist wohl klar, welcher der fünf Tests wahrscheinlich durchgefallen ist.
    - Sampling [Hoffman98]. Wir testen die wenigen Punkte, an denen wir die Antwort schon kennen, hier also z.B. Sin[π]=0. (Sollte mit anderen Methoden kombiniert werden, da die Aussagekraft gering ist.)
    - Metamorphes Testen [Chen98]. Bestandene Tests sind nicht nutzlos (im Sinne des Paradigmas, daß wir Fehler suchen) – wir können sie dazu benutzen, „metamorphe Regeln“ abzuleiten und damit neue sinnvolle Tests zu generieren. Es käme z.B. ein Test von „Sin(-X)=-Sin(X)“ zur Anwendung.
    - Paralleles Testen [Weyuker80]. Wir testen drei verschiedene Programme, von drei verschiedenen Programmierern. Alle drei liefern „239“. Es ist unwahrscheinlich, daß alle drei Programmierer denselben Fehler gemacht haben.
    - Maschinelles Lernen [Chan09]. Analog zu einem menschlichen Orakel versucht ein weiteres Programm, den Output des ersten Programms zu analysieren und darin nach Mustern und Strukturen zu suchen, aus deren Vorliegen es dann seine Antwort generiert.
    - Ducktesting. Analog zum Ducktyping [WikiDuck] wird getreu der Phrase “When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.” (aus einem Gedicht von J. W. Riley) angenommen, daß z.B. ein Programm, was 30 Jahre und 100 peer-reviewte Publikationen überstanden hat, die experimentellen Daten, die uns die Natur lieferte, wohl korrekt beschreiben sollte [INVEIG].

* Nicht-Orakel
    - Wir vertrauen den Fähigkeiten des Programmierers. Für „Wegwerf-Software“ ist dies auch völlig ausreichend. Übrigens liefert selbst das Nicht-Orakel noch 1 Bit Information: das Programm stürzt ab oder es läuft durch.

Die Anwendbarkeit und Angemessenheit eines Orakels ist immer eine praktische Entscheidung. Irgendein konkretes Orakel braucht immer noch gute, aussagekräftige Testfälle [Davis81]. Die genannten Orakel unterscheiden sich in vieler Hinsicht [Hoffman98]:

- Vollständigkeit
- Genauigkeit („239“ enthält mehr nutzbare Information als „ja“)
- Unabhängigkeit (von Programm und Umgebung)
- Geschwindigkeit (von Erstellung und Ausführung)
- Brauchbarkeit
- Änderbarkeit des Programms
- Nötiges Vorwissen

Letzterer Punkt [Kanewala13] ist bei wissenschaftlicher Software natürlich besonders relevant, wenn das Programm von einem „fremden“ Tester getestet werden soll.

3. Praxis

WIE?
Es sei nun der (Normal)Fall angenommen, daß wissenschaftliche Ergebnisse nicht im vornhinein bekannt sind und daher keine echten Orakel existieren. Alle oben genannten Pseudo-Orakel können jedoch grundsätzlich zur Anwendung kommen und werden teilweise kurz mit ausgewählten praktischen Beispielen illustriert.

3.1 **Paralleles Testen**

In [Davis81] wurde erstmals das Konzept vorgeschlagen (ohne spezifischen Namen – der ad-hoc-Name wurde vom Autor gewählt). Folgende Voraussetzungen sollten erfüllt werden, damit parallele Programme sinnvoll sind:

- Unabhängigkeit der Programme
- Schneller einfacher Compiler zum Schreiben des Pseudo-Orakel
- Programm ist für häufige Benutzung geplant
- Eindeutige Spezifikation des Programms
- Gute Testdaten für Input

(Letzterer Punkt ist natürlich nicht nur für diese Methode relevant, vgl. [Cox99] zum Thema Erstellen/Aufbereiten von guten Inputdaten.)
Eine offensichtliche Anwendung von parallelem Testen ist, eine neue Version eines Programms mit der alten zu testen (auch wenn dies natürlich gegen Punkt 1 verstößt und daher alte Fehler nicht findet – aber es ist ein sehr gutes Konzept, die neuen Fehler aufzudecken).
In [Feldt98] wurde dieser Gedanke noch einmal weiterentwickelt: die parallelen Programme wurden von einem genetischen Algorithmus generiert, was das aufwendige Schreiben der Programme überflüssig macht. Das Prinzip funktioniert, bei dem selbst gewählten Beispiel (Bremsen eines Flugzeugs per Seil) ist die Güte der Programme aber noch viel zu schlecht für einen Praxiseinsatz.
Ein Beispiel aus eigener Erfahrung möge dies Kapitel beschließen: Der Autor hatte ein einfaches Programm zur Berechnung von Dynkin-Indices [WikiDynkin] geschrieben, von dem er aber nicht wusste, ob die Ergebnisse korrekt waren. Ein Experte [Schellekens16] sandte ihm die Ergebnisse eines Vergleichsprogramm, die „garantiert richtig“ waren. Es zeigten sich unerklärliche Diskrepanzen. Nach vielem Suchen, wo denn der Bug versteckt war, erwies sich eine dritte Möglichkeit als zutreffend: beide Programme waren korrekt. Nach Jahrzehnten Benutzung war eine der Eingangsdateien des Kollegen durch einen Hardwarefehler korrumpiert – und hinterhältigerweise erzeugte sein Programm die meisten der Daten via Konsistenzregeln, so daß sowohl Heuristik- als auch Konsistenztests nicht anschlugen. (Diese Art von Hardwarefehler, die ein reiner Softwaretest nicht finden kann, ist in der Wissenschaft wegen der vielen Messgeräte ohnehin von größerer Bedeutung als in der reinen Informatik.)

3.2 **Metamorphes Testen (MT)**

In [Chen02] untersuchten die Autoren ein Programm zur numerischen Lösung partieller Differentialgleichungen. Für die üblichen Orakel bzw. Pseudo-Orakel (analytische Lösungen, Simulationen, Tabellen, Rechnung per Hand) war das Problem zu kompliziert, und Programmbibliotheken für eine parallele Rechnung waren nicht vorhanden (das Problem war zu speziell). Die Autoren bauten einen subtilen Fehler in das Programm ein, der bei flüchtigem Betrachten des Outputs kaum aufgefallen wäre. Allerdings lassen sich aus der Problemstellung gewisse Ungleichungen für benachbarte Werte auf dem Lösungsgitter ableiten, und diese wurden vom inkorrekten Programm verletzt. Hier zeigt sich ein Vorteil von MT: Metamorphe Regeln können nicht nur als Gleichungen formuliert werden, sondern auch als Fast-Gleichungen (wichtig beim Auftreten von Rundungsfehlern oder auch bei nichtdeterministischem Output) sowie als Ungleichungen.

[Murphy09] untersuchten die Vor- und Nachteile von MT: MT ist konzeptuell einfach und effektiv beim Fehlerfinden, aber aufwendig (sowohl bei der Aufbereitung von Input als auch von Output) und fehleranfällig in der Implementation (z.B. wirken sich kleine numerische Rundungsfehler, s.o., sehr lästig aus, indem sie echte Fehler vortäuschen). Sie demonstrierten, wie man mit einer Automatisierung von MT diese Nachteile beheben (und das praktische Potential dieser Methode natürlich enorm erhöhen) kann. Eine Automatisierung ist generell sowohl beim Erzeugen von neuen Testfällen (z.B. durch lineare Transformation, Umdrehen/Permutieren sowie Weglassen/Hinzufügen von Input) als auch beim Prüfen der metamorphen Regeln möglich. Die Autoren wendeten dies auf einige Programme auf dem Gebiet KI-Lernen an.

[Kanewala13] verglich MT, Runtime Assertions und KI-Lernen und fand, daß MT gegenüber Runtime Assertions Vorteile beim Finden von „Mutanten“ (künstlich erzeugten Fehlern, s. Punkt 2.1) aufwies. Hochgradig wünschenswert wäre, wenn auch das Finden von metamorphen Regeln automatisiert werden könnte.

Von Nutzen ist sicherlich auch, daß auch ein externer Tester ohne jegliche Kenntnis der internen Rechnungen eines Programms (z.B. ist die von [Murphy09] entworfene Testumgebung explizit zur Durchführung von Black-Box-Tests konzipiert) allein durch „trial and error“ bei systematischer Variation der Eingangsparameter durchaus eine gute Chance hat, metamorphe Regeln zu finden. (So gesehen kann man MT als eine spezielle Heuristik betrachten.)

3.3 **Statistik**

In [Mayer04] wurde dieses Verfahren für die Bildverarbeitung eingesetzt, bei der riesige Mengen an Daten anfallen, die man natürlich nicht mehr einzeln testen kann. Stattdessen wurden lediglich Mittelwerte auf ihre Korrektheit getestet. (Natürlich müssen für diese Werte vom Programm unabhängige Formeln existieren, und die Testfälle dürfen keine systematischen Verzerrungen enthalten.) Dieses einfache Konzept deckte nichtsdestotrotz verschiedene Arten von Fehlern auf: solche in Speicherzugriff, Verzweigungen, Operatoren, oder Objektzuständen. Problematisch waren Werte nahe 0 sowie durch Diskretisierung von Daten vorgetäuschte Fehler.

3.4 **KI-Lernen**

In [Chan09] wurde eine KI dazu benutzt, den Output von Mesh-Vereinfachungsalgorithmen zu testen. Wiederum ist die Datenmenge zu groß, um sie per Hand zu testen. Die KI wurde angelernt, indem ein groberes ähnliches, aber einfacheres Modell als Bewertungskriterium diente. Das Verfahren konnte allerdings nicht zur gleichen Zeit akkurat und robust sein.

3.5 **Ducktesting**

Hier kann der Autor auf 30 Jahre Erfahrung mit INVEIG [INVEIG] zurückgreifen, einem Programm zur Berechnung der Elektronenstrukturen von Seltenerdverbindungen.
Es folgt eine kurze Liste von Informationen über das Programm:

- Entwickelt am Lawrence Berkeley Laboratory
- Von Physikochemiker, nicht Informatiker
- Ca. 4000 Zeilen Fortran
- Davon etwa die Hälfte Kommentar (die interessanten Teile sind aber für einen Nichtspezialisten unverständlich)
- Intensive Nutzung von Standard-Bibliotheken (für Minimierung von parametrischen Zielfunktionen und Matrixmathematik)
- Zahlreiche Input-Flags, Benutzung auf eigene Gefahr
- Der Autor hat es auf einen Rechner der Uni Hamburg migriert sowie einige Zusatzprogramme geschrieben
- Ca. 100 peer-reviewte Publikationen basierend auf den Outputs

Im Rahmen dieser Arbeit liegen nun folgende Fragen nahe:
- Wurde INVEIG getestet?
Der Entwickler hat sicher getestet, was er konnte, aber ein Unit-Test der Unterprogramme im Sinne von „Dieser Steepest-Descent-Minimierer tut genau, was der Steepest-Descent-Algorithmus vorgibt“ wurde vermutlich nie durchgeführt (wozu auch, wenn man den Standard-Bibliotheken vertraut). Auch der Autor dieses Papers als Anwender hat etliche einfach vornehmbare Plausibilitätstests (z.B. „Kristallfeld Null heißt Multipletts spalten nicht auf“ – das ist ein Sampling-Test, oder „Kristallfeld negativ ist wie komplementäre f-Konfiguration“ – das ist essentiell ein metamorpher Test!) durchgeführt. Normalerweise würde er auch noch ein paar „Hardcore“-Tests machen (speziell mit den Input-Flags kann man das Programm schnell zum Absturz bringen), aber schon damals war das Programm gut bewährt (so daß er angesichts der mangelnden praktischen Relevanz keine Zeit verschwenden wollte), und heute interessiert ihn es nicht mehr wirklich.
- Muss man INVEIG weiter testen?
Das Ducktesting (die Ergebnisse des Programms stimmen wissenschaftlich anerkannt gut mit dem überein, wofür es geschrieben wurde, nämlich die Deutung von Spektren – so gesehen liefert die Natur selbst ein Testorakel) war erfolgreich.
- Kann (soll) man INVEIG weiter testen?
Nach 30 Jahren ist möglicherweise der Autor der letzte lebende Mensch, der gleichzeitig genug wissenschaftliche Expertise auf dem Gebiet aufweist, um überhaupt zu verstehen, was das Programm berechnet, und gleichzeitig auf dem Gebiet der Informatik hinreichend kompetent ist, um einen „echten“ Test zu designen. Eine saubere Re-Implementation von INVEIG in z.B. JAVA ohne obskure Flags, voll getestet und dokumentiert, ist zwar nicht unmöglich, aber bei einem (frei) geschätzten Aufwand von 100 Mannjahren schlicht absurd im Kosten/Nutzen-Verhältnis.

3.6. **Nicht-Orakel**

Der Autor ist Hobbymathematiker und programmiert ständig irgendwelche kleinen Programme zur Lösung von mathematischen Problemen. Mangels Relevanz sind sämtliche Fehler unkritisch und ein ernsthafter Test dieser Programme wäre reine Zeitverschwendung.

3.7. **Fazit**

Das Pseudo-Orakel für „untestbare“ (lies: ohne echtes Orakel) wissenschaftliche Programme gibt es noch immer nicht. Welches Verfahren man einsetzt, ist immer eine praktische, vom Kontext abhängige Entscheidung, und ohnehin ist das ganze Gebiet „Software testen“ noch im Fluss [Mili15]. Metamorphic Testing ist hier, wie unter Punkt 3.3 dargelegt, ein vielversprechendes Verfahren.

Unabhängig vom Verfahren ist Testautomatisierung, wo irgend möglich, sicherlich die Zukunft.

Nachwort

„Warum sparen wir nicht Zeit und testen einfach nur die Programmteile, die Fehler enthalten?“ (Unbekannter Witzbold, nach [Baumgartner13])

4. Literaturverzeichnis

a) Bücher: Testen generell

[Ammann08] P. Ammann, J. Offutt, „Introduction to Software Testing“, Cambridge University Press, Cambridge (Großbritannien), 2008.

[Baumgartner13] M. Baumgartner, M. Klonk, H. Pichler, R. Seidl, S. Tanczos, „Agile Testing“, Hanser, München, 2013.

[Binder00] R. V. Binder, „Testing Object-Oriented Systems: Models, Patterns and Tools“, Addison-Wesley, Boston (USA), 2000.

[Daigl16] M. Daigl, R. Glunz, „ISO 29119“, dpunkt.verlag, Heidelberg, 2016.

[Hendrickson14] E. Hendrickson, „Explore It!“, dpunkt.verlag, Heidelberg, 2014.

[Mili15] A. Mili, F. Tchier, „Software Testing“, Wiley, New Jersey (USA), 2015.

[Myers04] G. J. Myers, „The Art of Software Testing“, 2. Auflage, Wiley, New Jersey (USA), 2004.

[Seidl12] R. Seidl, M. Baumgartner, T. Bucsics, „Basiswissen Testautomatisierung“, dpunkt.verlag, Heidelberg, 1. Auflage, 2012.

[Sneed12] H. M. Sneed, M. Baumgartner, R. Seidl, „Der Systemtest“, Hanser, München, 3. Auflage, 2012.

[Spillner12] A. Spillner, T. Linz, „Basiswissen Softwaretest“, dpunkt.verlag, Heidelberg, 5. überarbeitete und aktualisierte Auflage, 2012.

b) Bücher, Artikel, Konferenzen: Testen speziell

[Chan09] W. K. Chan, S. C. Cheung, J. C. F. Ho, T. H. Tse, „PAT: A pattern classification approach to automatic reference oracles for the testing of mesh simplification programs“, Journal of Systems and Software **82**(3), 2009, pp. 422-434.

[Chen98] T. Y. Chen, S. C. Cheung, S. M. Yiu, „Metamorphic Testing: A New Approach for Generating Next Test Cases“, Technical Report HKUST-CS98-01, Hong Kong University of Science and Technology, 1998.

[Chen02] T. Y. Chen, J. Feng, and T. H. Tse, „Metamorphic testing of programs on partial differential equations: a case study.” Computer Software and Applications Conference (COMPSAC 2002), Proceedings. 26th Annual International. IEEE, 2002, pp. 327-333.

[Clune11] T. L. Clune, R. B. Rood, „Software testing and verification in climate model development“, Software, IEEE **28**(6), 2011, pp. 49-55.

[Cox99] M. G. Cox, P. M. Harris, „The design and use of reference data sets for testing scientific software“, Analytica Chimica Acta **380**(2), 1999, pp. 339-351.

[Davis81] Davis, Martin D., Elaine J. Weyuker, „Pseudo-oracles for non-testable programs.” Proceedings of the ACM'81 Conference, ACM, 1981, pp. 254-257.

[Feldt98] R. Feldt, „Generating diverse software versions with genetic programming: an experimental study.“ Software, IEE Proceedings, IET, **145**(6), 1998, pp. 228-236.

[Hoffman98] D. Hoffman, „A Taxonomy for Test Oracles”, Quality Week, **98**, 1998, pp. 52-60.

[Hoffman99] D. Hoffman, „Heuristic Test Oracles“, Software Testing & Quality Engineering Magazine, **1**(3-4), 1999, pp. 29-32.

[Kanewala13] U. Kanewala, J. M. Bieman, „Techniques for Testing Scientific Programs without an Oracle“, Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering, IEEE Press, 2013, pp. 48-57.

[Kelly10] D. Kelly, S. Thorsteinson, D. Hook, „Scientific Software Testing – Analysis in Four Dimensions“, IEEE Software, **28**(3), 2010, pp. 84-90.

[Mayer04] J. Mayer, R. Guderlei, „Test Oracles Using Statistical Methods”, in: Proceedings of the First International Workshop on Software Quality (SOQUA 2004), Ilmenau, Germany, September 2004, pp. 85-95.
[Murphy09] C. Murphy, K. Shen, G. Kaiser, „Automatic system testing of programs without test oracles“. In: Proceedings of the eighteenth international symposium on Software testing and analysis, ACM, 2009, pp. 189-200.
[Peters94] D. K. Peters, D. L. Parnas, „Generating a Test Oracle from Program Documentation“, Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA), ACM Press, 1994, pp. 58–65.
[Richardson92] D. J. Richardson, S. L. Aha, T. O. O'Malley, „Specification-based test oracles for reactive systems“, Proceedings of the 14th international conference on Software engineering, ACM, 1992, pp. 105-118.
[Robinson99] H. Robinson, „Finite State Model-Based Testing on a Shoestring”, Talk, STAR West, 1999.
[Staats11] M. Staats, M. W. Whalen, M. P. Heimdahl, „Programs, tests, and oracles: the foundations of testing revisited“, 2011 33rd International Conference on Software Engineering (ICSE), IEEE, 2011, pp. 391-400.
[Weyuker80] E. J. Weyuker, „The Oracle Assumption of Program Testing”, in: Proceedings of the 13th International Conference on System Sciences (ICSS), Honolulu, HI, January 1980, pp. 44-49.
[Whittaker00] J. A. Whittaker, „What is software testing? And why is it so hard?“, Software, IEEE **17**(1), 2000, pp. 70-79.

c) Artikel: Diverses
[Cristian85] F. Cristian, „A Rigorous Approach to Fault-Tolerant Programming”, IEEE Transactions on Software Engineering, **11**(1), 1985, pp. 23-31.
[Gödel31] K. Gödel, „Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I.“, Monatshefte für Mathematik und Physik, **38**, 1931, pp. 173-198.

d) Websites (letzter Test: Juni 2016)
[Bolton] http://www.developsense.com/blog/2012/05/oracles-cant-guarantee/
[MS] https://www.microsoft.com
[V] http://www.v-modell-xt.de
[Wiki6W] https://de.wikipedia.org/wiki/Nachricht_%28Journalismus%29#Stil_und_Inhalt
[WikiAriane] https://de.wikipedia.org/wiki/Ariane_V88
[WikiBug] https://en.wikipedia.org/wiki/Bebugging
[WikiDuck] https://de.wikipedia.org/wiki/Duck-Typing
[WikiDynkin] https://de.wikipedia.org/wiki/Dynkin-Index
[WikiOzon] https://en.wikipedia.org/wiki/Ozone_depletion#Antarctic_ozone_hole

e) Sonstiges, Fußnoten, Anmerkungen
[ISO9126] ISO/IEC 9126, International Standard for the Evaluation of Software Quality.
[LB] Diese Programmierweisheiten sind so alt, daß die Herkunft nicht mehr feststellbar ist. Vgl. auch den satirischen Webcomic „Lovelace and Babbage“, http://sydneypadua.com/2dgoggles/
[INVEIG] Lawrence Berkeley Laboratory, Programmierer und Datum nicht bekannt (ca. 1960).
[Schellekens16] Prof. Bert Schellekens (NIKHEF, Niederlande), persönliche Mitteilung, 2016.