

Delta Debugging

Yesterday, my program worked.
Today, it does not. Why?

Seminararbeit von
Alexander Gitter

im Rahmen des Seminars
„Software-Qualitätssicherung: Testen, Debuggen, Verifizieren“
im Wintersemester 2007/2008

Betreuer:
Prof. Dr. W. Küchlin
Dipl.-Inform. H. Post
Dr. C. Sinz

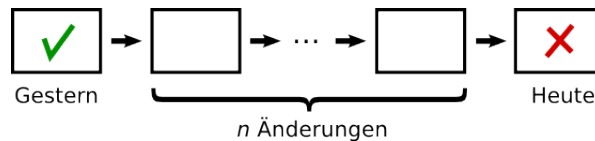
Inhaltsverzeichnis

1	Einleitung	3
2	Manuelle Fehlersuche	3
3	Automatisierte Fehlersuche	4
4	Delta Debugging	5
4.1	Überblick	5
4.2	Erster Ansatz	5
4.3	Behandlung von Inkonsistenzen	7
5	Beispiel	8
6	Zusammenfassung und Ausblick	11
A	Quellcode	12
	Literatur	17

1 Einleitung

Bei der Entwicklung von Software ist es oftmals der Fall, dass das Umstrukturieren oder Einfügen von neuem Code, Auswirkungen auf bereits vorhandene und funktionierende Teile hat. So kommt es vor, dass Programmteile plötzlich nicht mehr funktionieren – die Software verweigert ihren Dienst oder vorher bestandene Testfälle zeigen einen Fehler an.

Ist die alte, fehlerfreie Version (*Gestern*) bekannt, sowie alle Änderungen, die zur aktuellen, fehlerbehafteten Version (*Heute*) des Codes führten, so lässt sich ein Verfahren entwickeln, welches komplett automatisch die zum Fehlverhalten führende Änderung ausfindig macht. Die Problemstelle im Quellcode wird dann entweder direkt aufgezeigt oder aber zumindest die zu untersuchenden Stellen stark eingeschränkt.



Im Folgenden wird vor allem auf Programmcode Bezug genommen. Eine Änderung ist hierbei nichts weiter als ein „Diff“ zweier Versionen des Quellcodes. Das Verfahren ist jedoch nicht auf Quelltext beschränkt. Allgemein können alle solche Dinge betrachtet werden, wo sich die Unterschiede zwischen zwei Versionen in irgendeiner Weise als Delta darstellen lassen. Beispiele hierfür sind Konfigurationseinstellungen, Threadablaufplanungen oder Ähnliches.

Dieser Ausarbeitung, sowie dem Seminarvortrag, liegen vor allem die Quellen [1, 2] zugrunde.

2 Manuelle Fehlersuche

Um einen Fehler in einem Programmcode ausfindig zu machen, bietet sich oftmals die folgende Vorgehensweise an.

1. Fehler reproduzieren und automatischen Testfall erzeugen

Dieser Schritt stellt die verlässliche Reproduzierbarkeit eines Programmfehlers sicher. Das ist eine wichtige Voraussetzung um im Weiteren logische Schlüsse ziehen zu können. Außerdem lässt sich so die Wirksamkeit einer späteren Fehlerkorrektur bestätigen.

2. Isolierung der Fehlerursache

Nun muss die eigentliche Fehlerursache isoliert werden. Hierbei handelt es sich um die Suche nach Code, der das Programm in einen ungünstigen Zustand überführt. Im weiteren Verlauf kommt es schließlich zu dem vorher beobachteten, charakteristischen Fehlverhalten. Werkzeuge wie Debugger sind in diesem Schritt eine wichtige Hilfe.

Es ist sinnvoll weitere bekannte Daten heranzuziehen – etwa alle Änderungen an der Software zwischen *Gestern* und *Heute*.

3. Fehler beheben und erneutes Testen

Wenn die Fehlerursache bekannt ist, kann der Programmcode so abgeändert werden, dass ein Übergang in den fehlerhaften Zustand nicht mehr auftritt. Danach muss überprüft werden, ob das Problem tatsächlich behoben ist. Dies kann zum Beispiel durch erneutes Auswerten der in Schritt 1 erstellten Testfälle passieren.

Probleme dieser Methode

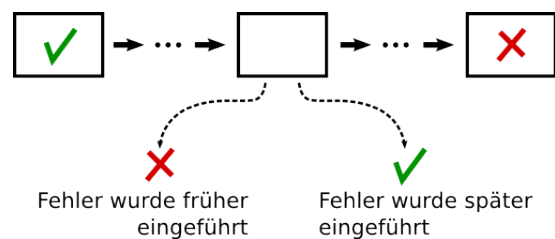
Die manuelle Fehlersuche kann sehr komplex sein. So müssen bei der Verwendung eines Debuggers typischerweise zunächst geeignete Stellen für Breakpoints ausgewählt werden. Fällt erst im Nachhinein auf, dass für die Analyse wichtige Codestellen übersprungen wurden, so werden weitere Breakpoints gesetzt und der Debuggingvorgang muss erneut gestartet werden.

Dieses Einkreisen der Fehlerursache ist zeitaufwändig. Erschwerend kommt hinzu, dass auch in kurzen Zeiträumen viele Änderungen am Code durchgeführt werden können und Programmierer parallel an unterschiedlichen Programmteilen arbeiten.

Im Folgenden werden Methoden vorgestellt, die Fehlerursachen im Quellcode so weit wie möglich automatisch auffinden können.

3 Automatisierte Fehlersuche

Ein erster Ansatz zum automatischen Finden eines Fehlers basiert auf der binären Suche. Dazu wird das Programm auf sein Verhalten in verschiedenen Versionen zwischen *Heute* und *Gestern* untersucht. Ist der aktuell getestete Zustand schlecht, so wurde ein Fehler mit diesem oder einem früheren Zustand eingeführt. Ist der Zustand in Ordnung, so wird mit einem späteren Zustand weiter verfahren.



Wie bei der binären Suche kann in jedem Schritt mit dem Zustand fortgefahren werden, der mittig im aktuell betrachteten Intervall liegt. *Gestern* und *Heute* sind als „gut“ bzw. „schlecht“ bekannt. Erzeugt ein Zustand x den Fehler, der Zustand $x - 1$ jedoch nicht, dann terminiert das Verfahren und man kann davon ausgehen dass das Problem in Zustand x eingeführt wurde.

Probleme dieser Methode

- **Interferenz**
Wenn mehrere Änderungen einen Fehler nur in Kombination verursachen, dann liegt eine Interferenz vor.
- **Granularität**
Eine logische Änderung kann viele Zeilen Code umfassen. Wenn eine bestimmte Änderung als problematisch erkannt ist, bleibt trotzdem noch einige manuelle Arbeit um das wesentliche Codestück zu isolieren.
Wird die Änderungsgröße andererseits zu feingranular gewählt, stößt man vor allem auf das im nächsten Punkt beschriebene Problem der Inkonsistenz.

- **Inkonsistenz**

Teilweise Anwendung von Änderungen kann zu ProgramMZuständen führen, die sich mit den angelegten automatischen Tests nicht mehr testen lassen. So ist ein Quellcode mit fehlerhafter Syntax nicht kompilierbar und damit keine Aussage über das Verhalten zur Laufzeit möglich.

4 Delta Debugging

4.1 Überblick

Delta Debugging ist ein automatisches Verfahren zum Auffinden der Änderung, die einen Fehler verursacht. Dabei ist der allgemeine Algorithmus nicht auf Änderungen einer bestimmten Form, etwa Codeänderungen, beschränkt. Viel mehr ist Delta Debugging auch anwendbar auf Änderungen von Eingabedaten, Systemkonfigurationen, Thread-Ablaufplanung oder anderen Parametern die das Laufzeitverhalten eines Programmes beeinflussen können. Inkonsistenzen werden sinnvoll behandelt, um auch feingranulare Änderungen unterstützen zu können. Interferenzen werden in linearer Zeit gefunden, für Einzelfehler ist die Laufzeit logarithmisch.

Konfigurationen

Eine *Konfiguration* c ist eine Teilmenge aller vorgenommenen Änderungen:

$$c \subseteq \{\Delta_1, \Delta_2, \dots, \Delta_n\}.$$

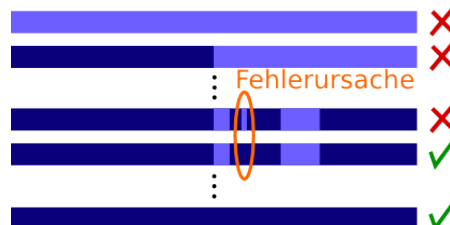
In der Praxis hat eine solche Teilmenge von Änderungen meist zwei wichtige Eigenschaften:

- Erzeugt eine Änderung einen Fehler, so wird auch jede Konfiguration, die diese Änderung enthält, den Fehler erzeugen (*Monotonie*)
- Ein bestimmter Fehler wird von nur einer Änderungsmenge erzeugt, nicht von mehreren unterschiedlichen (*Eindeutigkeit*).

Außerdem werden wird zunächst auch die Konsistenzeigenschaft angenommen, d.h. die Anwendung beliebiger Teilmengen von Änderungen führt immer in einen Konsistenten Zustand des Programmcodes. Inkonsistent wird der Code dann, wenn eine Änderungsmenge beispielsweise einen Syntaxfehler verursacht. In diesem Fall könnte der Testfall nicht ausgewertet werden.

4.2 Erster Ansatz

Anhand dieser Eigenschaften kann eine Fehlerursache durch wiederholtes Aufteilen der Konfiguration *Heute* gefunden werden. Trifft man auf eine einzelne Änderung, die zwischen Bestehen und Nicht-Bestehen des Testfalles entscheidet, dann muss es dieser Code sein, der für den Fehler verantwortlich ist.



An einem Beispiel soll dies demonstriert werden. Die Menge aller Änderungen ist hier durch die Zahlen 1 bis 8 repräsentiert. Eine Zahl in der Spalte Konfiguration zeigt an, dass diese Änderung im jeweiligen Schritt in der Konfiguration enthalten ist. Die Spalte c gibt die Bezeichnung der Konfiguration im jeweiligen Schritt an. Die Schritte 1 und 2 testen beispielsweise zwei Hälften (c_1 und c_2) der gesamten Änderungsmenge.

Nach den Schritten 1 und 2 ist klar, dass der Fehler nur durch eine Kombination von Änderungen aus den unterschiedlichen Konfigurationen c_1 und c_2 hervorgerufen wird (Interferenz). Daher wird im nächsten Schritt c_2 festgehalten und $c \leftarrow c_1$ in zwei weitere Hälften geteilt. Der Prozess beginnt von vorn, indem diese beiden neuen Teile (wiederum mit c_1 und c_2 bezeichnet) untersucht werden. Nach Schritt 5 steht die Beteiligung von Änderung 3 fest, das gleiche Vorgehen mit Vertauschen der ursprünglichen c_1 und c_2 stellt Änderung 6 als zweite Notwendigkeit heraus.

Schritt	c	Konfiguration	Test
1	c_1	1 2 3 4	✓
2	c_2 5 6 7 8	✓
3	c_1	1 2 . . 5 6 7 8	✓
4	c_2	. . 3 4 5 6 7 8	✗
5	c_1	. . 3 . 5 6 7 8	✗
6	c_1	1 2 3 4 5 6 . .	✗
7	c_1	1 2 3 4 5 . . .	✓
Ergebnis		. . 3 . . 6 . .	

Man beachte, dass hier die Eigenschaft der Eindeutigkeit vorausgesetzt wird. Würden die Änderungen 4 und 8 in Kombination den Fehler ebenso auslösen, so könnte man dies in einem einzelnen Durchlauf nicht herausfinden.

Ganz allgemein lässt sich der Algorithmus wie folgt beschreiben:

1. Teile Konfiguration c in zwei Teile c_1 und c_2
2. Wenn Test c_1 fehlschlägt, dann enthält c_1 die fehlerbehaftete Änderung
 \Rightarrow im Folgenden muss c_1 genauer betrachtet werden, während c_2 nicht weiter interessiert
(Schlägt c_2 fehl, dann analog)
3. Sind beide Tests erfolgreich, so muss das Fehlverhalten durch eine Kombination von Änderungen aus c_1 und c_2 hervorgerufen werden
 \Rightarrow Halte eine der beiden Teile fest und beginne mit dem anderen von vorn

Eigenschaften

Dieser erste Algorithmus hat eine lineare Worst-Case Laufzeit. Sollte eine Konfiguration mehrdeutig oder nicht monoton sein, so liefert der Algorithmus

- bei Interferenzen eine fehlerverursachende Änderungsmenge, die evtl. nicht minimal ist, sowie
- für Einzelfehler das richtige Ergebnis bei Nicht-Monotonie, bzw. nur eine der mehreren fehlerverursachenden Änderungen bei Mehrdeutigkeit.

Liegt die zweite Situation vor, so können die fehlerhaften Stellen nacheinander korrigiert und der Algorithmus nach jeder einzelnen Verbesserung erneut angewendet werden.

4.3 Behandlung von Inkonsistenzen

Zunächst wurden hier nur die Fälle ✓ und ✗ behandelt. Das wichtigste Problem bei der willkürlichen Anwendung von Änderungen – Inkonsistenz – wurde bisher ausgeklammert. So kommt es zum Beispiel häufig vor, dass der zu testende Code aufgrund eines Syntaxfehlers nicht kompilierbar ist. Es soll deshalb ein dritter Testausgang eingeführt werden: ?, Testfall nicht auswertbar.

Im schlimmsten Fall sind alle Tests inkonsistent. Beispielsweise haben im weiter unten stehenden Diagramm die ersten Konfigurationen c_1 und c_2 beide den Testausgang ?. Um wieder zu einer konsistenten Konfiguration zu kommen, lässt sich die Tatsache ausnutzen, dass die Konfigurationen *Gestern* und *Heute* immer konsistent sind.



Es bietet sich somit an, zu einer inkonsistenten Konfiguration weitere Änderungen hinzuzufügen (Annäherung an *Heute*) bzw. Änderungen wegzunehmen (Annäherung an *Gestern*). Da eine Konfiguration bisher am Anfang in nur zwei Teile zerlegt wurde, ist dies allerdings nicht möglich ohne die komplette Konfiguration wieder herzustellen. Daher wird im erweiterten Algorithmus eine Aufteilung in n Teilmengen vorgenommen, abhängig vom Ausgang vorangegangener Durchläufe.

1. Teile c in n Teilmengen (zu Beginn $n = 2$) und teste c_i bzw. \bar{c}_i
2. Behandle die Testergebnisse wie folgt:

Fehler

Beginne mit c_i von vorn, wenn $\text{test}(c_i) = \text{✗}$

Interferenz

Betrachte c_i mit festem \bar{c}_i und \bar{c}_i mit festem c_i , wenn $\text{test}(c_i) = \text{test}(\bar{c}_i) = \text{✓}$

Präferenz

Betrachte c_i mit festem \bar{c}_i , wenn $\text{test}(c_i) = \text{?}$ und $\text{test}(\bar{c}_i) = \text{✓}$

Erneuter Versuch

In allen anderen Fällen, wiederhole mit $n \leftarrow 2n$

Im folgenden Beispiel ist Änderung 8 fehlerinduzierend. Die Änderungen 2, 3 und 7 implizieren sich gegenseitig, das heißt sie müssen immer gemeinsam auftauchen, andernfalls liegt ein inkonsistenter Zustand vor.

Schritt	c_i	Konfiguration	Test	
1	$c_1 = \overline{c_2}$	1 2 3 4	?	
2	$c_2 = \overline{c_1}$ 5 6 7 8	?	
3	c_1	1 2	?	Erneuter Versuch (teile in 2n Teilmengen)
4	c_2	. . 3 4	?	
5	c_3 5 6 . .	✓	
6	c_4 7 8	?	
7	$\overline{c_1}$. . 3 4 5 6 7 8	?	Teste Komplemente der Schritte 3 bis 6
8	$\overline{c_2}$	1 2 . . 5 6 7 8	?	
9	$\overline{c_3}$	1 2 3 4 . . 7 8	✗	
10	$\overline{c_4}$	1 2 3 4 5 6 . .	?	
11	c_1	1 . . . 5 6 . .	✓	Erneuter Versuch / Präferenz (5, 6 bleiben fest)
12	c_2	. 2 . . 5 6 . .	?	
13	c_3	. . 3 . 5 6 . .	?	
14	c_4	. . . 4 5 6 . .	✓	
15	c_5 5 6 7 .	?	
16	c_6 5 6 . 8	✗	
Ergebnis	 8		

Im Allgemeinen kann eine verbesserte Behandlung von Inkonsistenzen wesentlich zur Effizienz des Algorithmus beitragen. So könnten beispielsweise verwandte Änderungen gruppiert werden, welche dann möglichst lange gemeinsam in den aufeinanderfolgenden Konfigurationen auftauchen sollten. Kriterien wären hier etwa chronologische, räumliche oder syntaktische Zusammengehörigkeit.

Aufgrund solcher Eigenschaften könnte man für bestimmte Konfigurationen auch eine Inkonsistenz vorhersagen und so den eigentlichen Testlauf einsparen.

5 Beispiel

Anhand mehrerer kleiner Beispiele soll demonstriert werden, welche Ergebnisse der Delta-Debugging-Algorithmus liefern kann.

Als Grundlage wurde eine Python-Implementierung des DD-Algorithmus [3] verwendet. Darauf baut ein, ebenfalls in Python implementierter, Testfall auf, welcher die Funktionsweise eines einfachen C-Programmes auf Richtigkeit überprüft (Anhang A, Listing 4).

Das zu testende Programm übernimmt beliebig viele, ganze Zahlen als Kommandozeilenargumente und soll diese, in sortierter Reihenfolge, wieder ausgeben. Zwei Aufrufe der korrekten und fehlerhaften Varianten könnten etwa wie folgt aussehen:

```
$ ./sort_gestern 14 3
Output: 3 14
$ ./sort_heute 14 3
Output: 0 3
```

Folgende Codeausschnitte aus beiden Varianten zeigen, dass es einige Änderungen gab.

Gestern

```

void shell_sort(int a[], int size)
{ ... }

int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *) malloc((argc-1)
                      * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc - 1);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");
}

```

Heute

```

void shell_sort(int a[], int size)
{ ... }

int main(int argc, char** argumente)
{
    int *a;
    int i;
    int outcount = argc;

    a = (int *) malloc(argc * sizeof(int)
                      - sizeof(int));
    for (i = 1; i < argc; i++)
        a[i - 1] = atoi(argumente[i]);

    shell_sort(a, argc--);

    printf("Output: ");
    for (i = 1; i < outcount; i++)
        printf("%d ", a[i - 1]);
    printf("\n");
}

```

Wird nun ein entsprechender Testfall entworfen, so liefert die einfache Implementierung des DD-Algorithmus folgende Ausgabe. Wichtig ist die hervorgehobene Zeile, die angibt, welche Änderung den Fehler hervorruft.

```

* Hier ist der Einsprungspunkt des Programmes.
*/
int main(int argc, char **argumente)
.
=== Status:
0
Output: 3 10
=== PASS ===

dd: 1 deltas left: [(2, '\n34c\n  shell_sort(a, argc--);\n.\n')]
dd: done
[The 1-minimal failure-inducing difference is [(2, '\n34c\n  shell_sort(a, argc--);\n.\n')]
[(3, '\n30,32c\n  a = (int *)malloc((argc) * sizeof(int) - sizeof(int));\n  for (i = 1; i <
argc; i++)\n    a[i - 1] = atoi(argumente[i]);\n.\n'), (4, '\n28a\n  int outcount = argc
;\n.\n'), (5, '\n25c\n/*\n * Die main-Methode.\n * Hier ist der Einsprungspunkt des Programmes.
\n */\nint main(int argc, char **argumente)\n.\n'), (1, '37,38c\n  for (i = 1; i < outcount;
i++)\n    printf("%d ", a[i-1]);\n.\n')] passes, [(1, '37,38c\n  for (i = 1; i < outcount
; i++)\n    printf("%d ", a[i-1]);\n.\n'), (2, '\n34c\n  shell_sort(a, argc--);\n.\n'), (
3, '\n30,32c\n  a = (int *)malloc((argc) * sizeof(int) - sizeof(int));\n  for (i = 1; i < a
rgc; i++)\n    a[i - 1] = atoi(argumente[i]);\n.\n'), (4, '\n28a\n  int outcount = argc;\n
.\n'), (5, '\n25c\n/*\n * Die main-Methode.\n * Hier ist der Einsprungspunkt des Programmes.\n
*/\nint main(int argc, char **argumente)\n.\n')] fails

```

Die dort angegebene Änderung ist hier noch einmal etwas besser dargestellt:

<pre>main(int argc, char *argv[]) int *a; int i; a = (int *)malloc((argc - 1) * sizeof(int)); for (i = 0; i < argc - 1; i++) a[i] = atoi(argv[i + 1]); shell_sort(a, argc-1); printf("Output: "); for (i = 0; i < argc - 1; i++) printf("%d ", a[i]); printf("\n"); free(a);</pre>	<pre>/* * Die main-Funktion. * Hier ist der Einsprungspunkt des Prog */ int main(int argc, char **argumente) { int *array; int i; int outcount = argc; array = (int *)malloc((argc) * sizeof(int)); for (i = 1; i < argc; i++) array[i - 1] = atoi(argumente[i]); shell_sort(array, argc-1); printf("Output: "); for (i = 1; i < outcount; i++) printf("%d ", array[i-1]); printf("\n"); free(array);</pre>
---	--

Natürlich ändert diese Modifizierung des Ausdruckes `argc-1` auf die Anweisung `argc--` die Semantik des Programms. Ein Blick in den Code (siehe Anhang) verrät, dass tatsächlich diese Zeile für die fehlerhafte Ausgabe verantwortlich ist. So wurde vorher der um eins verringerte Wert von `argc` an die Sortierfunktion übergeben – durch das Postinkrement bleibt es beim ursprünglichen Wert. Die Funktion arbeitet nun mit dem Array, als hätte dies ein Element mehr als es wirklich der Fall ist.

Der DD-Algorithmus hat hier also unter vielen Änderungen diejenige Zeile herausgefunden, welche das Fehlverhalten des Programms verursacht.

Als nächstes sind mehrere Änderungen gemacht worden, die Abhängigkeiten in verschiedenen Zeilen nach sich ziehen (z.B. das Umbenennen einer Variable). Die ursprüngliche Fehlerursache liegt hierbei aber in der gleichen Änderung wie oben.

<pre>main(int argc, char *argv[]) int *a; int i; a = (int *)malloc((argc - 1) * sizeof(int)); for (i = 0; i < argc - 1; i++) a[i] = atoi(argv[i + 1]); shell_sort(a, argc-1); printf("Output: "); for (i = 0; i < argc - 1; i++) printf("%d ", a[i]); printf("\n"); free(a);</pre>	<pre>/* * Die main-Funktion. * Hier ist der Einsprungspunkt des Prog */ int main(int argc, char **argumente) { int *array; int i; int outcount = argc; array = (int *)malloc((argc) * sizeof(int)); for (i = 1; i < argc; i++) array[i - 1] = atoi(argumente[i]); shell_sort(array, argc-1); printf("Output: "); for (i = 1; i < outcount; i++) printf("%d ", array[i-1]); printf("\n"); free(array);</pre>
---	--

Delta Debugging bringt hier folgende Ausgabe

```
The 1-minimal failure-inducing difference is [(1, '41c\n    free(array);\n.\n'), (2, '\n37,38c\n    for (i = 1; i < outcount; i++)\n        printf("%d ", array[i-1]);\n.\n'), (3, '\n34c\n    shell_sort(array, argc-);\n.\n'), (4, '\n30,32c\n        array = (int *)malloc((argc) * sizeof(int) - sizeof(int));\n        for (i = 1; i < argc; i++)\n            array[i - 1] = atoi(argumente[i]);\n.\n'), (6, '\n27c\n        int *array;\n.\n'), (7, '\n25c\n/*\n * Die main-Methode.\n * Hier ist der Einsprungspunkt des Programmes.\n */\nint main(int argc, char **argumente)\n.\n')]
```

die man zum einfacheren Lesen wie folgt darstellen kann

```
- int main(int argc, char *argv[])
+ int main(int argc, char **argumente)

-     int *a;
+     int *array;
```

```
-     for (i = 0; i < argc - 1; i++)
-         a[i] = atoi(argv[i + 1]);
+     for (i = 1; i < argc; i++)
+         array[i - 1] = atoi(argumente[i]);

-     shell_sort(a, argc - 1);
+     shell_sort(array, argc--);
```

Das Programm hat hier mit Hilfe des Delta Debugging Algorithmus also wieder die fehlerverursachende Zeile identifiziert. Aufgrund der Umbenennung von Variablen, gibt es hier allerdings Inkonsistenzen wenn versucht wird den Code mit nur einem Teil der voneinander abhängigen Änderungen zu kompilieren. Das Verfahren muss diese Änderungen somit auch in die Ergebnismenge übernehmen – den eigentlichen Fehler zu finden ist hier damit nicht so offensichtlich wie im ersten Fall.

6 Zusammenfassung und Ausblick

Delta Debugging ist eine effektive Methode zum Auffinden von fehlerinduzierenden Änderungen in einem Quelltext. Hierbei ist kein Benutzereingriff nötig und der Algorithmus kann mit komplizierten Details eines Testlaufs umgehen. Dies ermöglicht interessante Anwendungen, wie die Integration in Regressionstests, wodurch Entwickler direkt wichtige Anhaltspunkte für das Beheben eines Problems erhalten können.

Zur Vermeidung von Inkonsistenzen könnten Änderungen bezüglich verschiedener Kriterien (zum Beispiel bei chronologischer, räumlicher oder syntaktischer Verwandtschaft) gruppiert werden. Hält man diese Änderungen so lange wie möglich zusammen, können Inkonsistenzen vermieden, und so der Ablauf effizienter gestaltet werden.

A Quellcode

Listing 1: Beispielprogramm im Zustand *Gestern*

```

#include <stdio.h>
#include <stdlib.h>
static void shell_sort(int a[], int size) {
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);

    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

int main(int argc, char *argv[]) {
    int *a;
    int i;
    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc-1);

    printf("Output:␣");
    for (i = 0; i < argc - 1; i++)
        printf("%d␣", a[i]);
    printf("\n");

    free(a);
    return 0;
}

```

Listing 2: Beispielprogramm im Zustand *Heute*

```

#include <stdio.h>
#include <stdlib.h>
/* shell_sort:
 * sortiert ein Array a der Laenge size aufsteigend
 */
static void shell_sort(int a[], int size) {
    int x, var;
    int h = 1;

```

```

do {
    h = h * 3 + 1;
} while (h <= size);
do {
    h /= 3;
    for (x = h; x < size; x++)
    {
        int v = a[x];
        for (var = x; var >= h && a[var - h] > v; var -= h)
            a[var] = a[var - h];
        if (x != var)
            a[var] = v;
    }
} while (h != 1);
}

/*
 * Die main-Funktion.
 * Hier ist der Einsprungspunkt des Programmes.
 */
int main(int argc, char **argumente) {
    int *a;
    int i;
    int outcount = argc;

    a = (int *)malloc((argc) * sizeof(int) - sizeof(int));
    for (i = 1; i < argc; i++)
        a[i - 1] = atoi(argumente[i]);

    shell_sort(a, argc--);

    printf("Output: ");
    for (i = 1; i < outcount; i++)
        printf("%d ", a[i-1]);    printf("\n");

    free(a);
    return 0;
}

```

Listing 3: Beispielprogramm 2 im Zustand *Heute*

```

#include <stdio.h>
#include <stdlib.h>
/* shell_sort:
 * sortiert ein Array a der Laenge size aufsteigend
 */
static void shell_sort(int a[], int size) {
    int i, j;
    for (i = 0; i < size; ++i) {
        for (j = 0; j < size - i - 1; ++j) {
            if (a[j] > a[j + 1]) {
                int tmp = a[j];
                a[j] = a[j + 1];
            }
        }
    }
}

```

```

        a[j + 1] = tmp;
    }
}
}
/*
 * Die main-Funktion.
 * Hier ist der Einsprungspunkt des Programmes.
 */
int main(int argc, char **argumente) {
    int *array;
    int i;
    int outcount = argc;

    array = (int *)malloc((argc) * sizeof(int) - sizeof(int));
    for (i = 1; i < argc; i++)
        array[i - 1] = atoi(argumente[i]);

    shell_sort(array, argc--);

    printf("Output: ");
    for (i = 1; i < outcount; i++)
        printf("%d ", array[i-1]);
    printf("\n");

    free(array);

    return 0;
}

```

Listing 4: Testfall, der den DD-Algorithmus benutzt

```

import DD
import commands
import os

class SampleDD(DD.DD):
    def __init__(self, deltacount):
        DD.DD.__init__(self)
        self.deltacount = deltacount

    def applyChange(self, filename, edcmds):
        out = open("tmp.ed", "w")
        out.write(edcmds.strip())
        out.write("\n1,$p\n")
        out.close()
        cmd = "ed -s " + filename + " < tmp.ed 2> /dev/null"
        (status, output) = commands.getstatusoutput(cmd)

        return output

    def _test(self, deltas):
        # Build input
        print "===== Starting Test ====="

```

```
try:
    os.remove("a.out")
except:
    pass

count = 0

tabelle = ( "." * self.deltacount ) + " "

for(index, delta) in deltas:
    tabelle = tabelle[: (index-1)] + str(index)
                                     + tabelle[ (index-1)+1:]

    if(count == 0):
        newversion = self.applyChange("s1.c", delta)
    else:
        newversion = self.applyChange("tmp.c", delta)

    print "=====" + " Applying " + str(index) + ":" + delta

    out = open("tmp.c", "w")
    out.write(newversion)
    out.close()
    count = count + 1

(status, output) = commands.getstatusoutput("gcc tmp.c")
print "=== Status: "
print status

tabout = open("tabelle.txt", "a")

if status != 0:
    print "=== UNRESOLVED ==="
    tabout.write(tabelle + " U\n")
    tabout.close()
    return self.UNRESOLVED

output = commands.getoutput("./a.out 10 3")
print output

if(output.strip() == "Output: 3 10"):
    print "=== PASS ==="
    tabout.write(tabelle + " P\n")
    tabout.close()
    return self.PASS
else:
    print "=== FAIL ==="
    tabout.write(tabelle + " F\n")
    tabout.close()
    return self.FAIL

if __name__ == '__main__':
    deltas = []
```

```
index = 1
cmd = "diff -e s1.c s3.c | awk -f ed.awk"
(status, output) = commands.getstatusoutput(cmd)

try:
    os.remove("tabelle.txt")
except:
    pass

for delta in output.split("==="):
    deltas.append((index, delta))
    index = index + 1

sampledd = SampleDD(len(deltas))

print "Isolating the failure-inducing difference..."
(c, c1, c2) = sampledd.dd(deltas)
print "The 1-minimal failure-inducing difference is", c
print c1, "passes,", c2, "fails"
```

Literatur

- [1] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proc. of the 7th European Software Engineering Conf./7th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE'99) (Toulouse, France, 1999)*, 253–267.
- [2] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Dpunkt Verlag, 2005. ISBN 3-89864-279-8
- [3] Software Engineering Chair (Prof. Zeller) - Saarland University. *Delta Debugging*. URL: <http://www.st.cs.uni-sb.de/dd/> (Abruf am 17. Januar 2008)