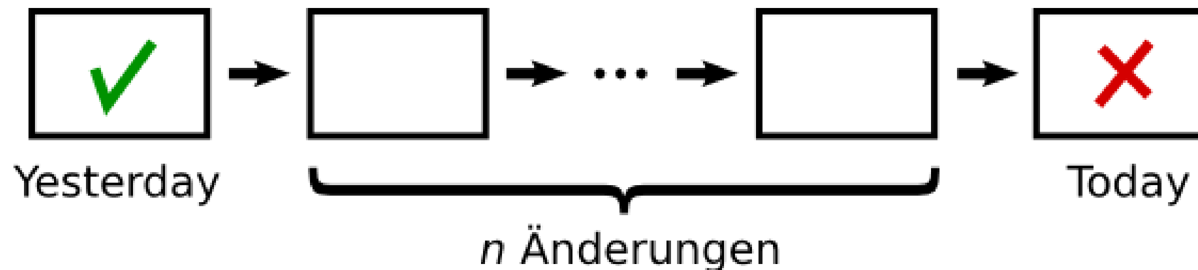
Delta Debugging

Alexander Gitter

Betreuer: Carsten Sinz

Einleitung

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



Einleitung

Änderung

- Beschreibt den Unterschied („Delta“) zwischen einem Zustand und einem anderen Zustand
  - Für Programmcode lassen sich Änderungen beispielsweise als „diff“ darstellen

Manuelle Fehlersuche

- Fehler reproduzieren
- Automatischen Test aufsetzen

Einfacher Ansatz

- Programm per Hand debuggen

Hilfreich

- Nach der Änderung suchen, die den Fehler verursacht

Probleme

- Schon ein einziger Programmierer kann an einem Tag viele Änderungen vornehmen
- Paralleles Arbeiten an unterschiedlichen Programmteilen
- Einkreisen des Fehlers im Code ist sehr zeitaufwändig
  - Typischerweise durch Verwendung eines Debuggers
    - Wo sollen Breakpoints gesetzt werden?
    - „I stepped too far“

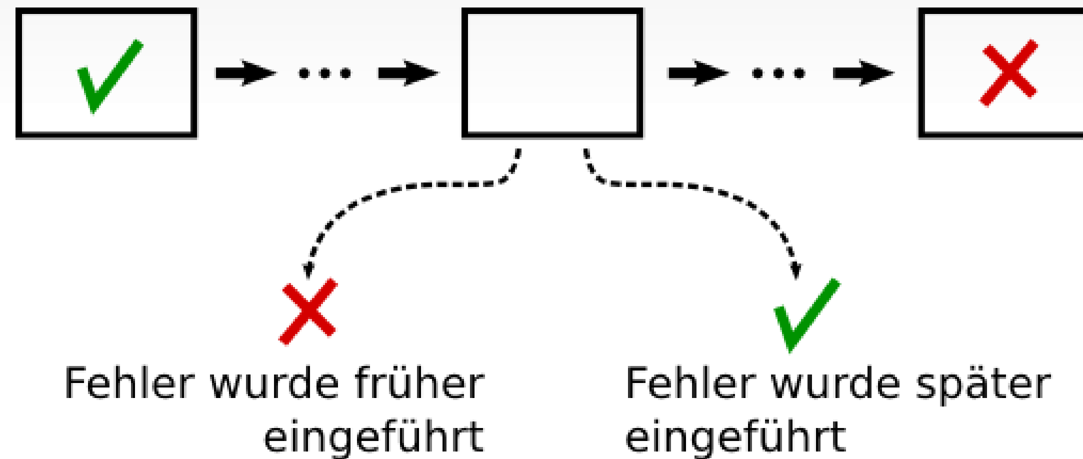
Probleme

Linux 2.6.24

- 9836 Patches, ~1,2 Mio Änderungen an 10209 Dateien
- 950 Entwickler

Automatisierte Methode

Binäre Suche



Probleme

Interferenz

- Änderungen verursachen einen Fehler nur in Kombination

Granularität

- Eine logische Änderung umfasst viele Zeilen Programmcode

Inkonsistenz

- Teilweise Anwendung von Änderungen führt zu inkonsistentem Programmcode

Delta Debugging

Überblick

- Automatisches Verfahren
- Simplifizierung
  - Finde die kleinste Eingabe, die einen Fehler verursacht
- Isolierung
  - Finde die Änderung, welche den Fehler verursacht
- Anwendbar auf Eingabedaten, **Codeänderungen**, Thread-Ablaufplanung, ...

Delta Debugging

- Behandelt Inkonsistenzen sinnvoll um feingranulare Änderungen zu unterstützen
- Findet beliebige Interferenzen in linearer Zeit
- Findet Einzelfehler in logarithmischer Zeit

Delta Debugging

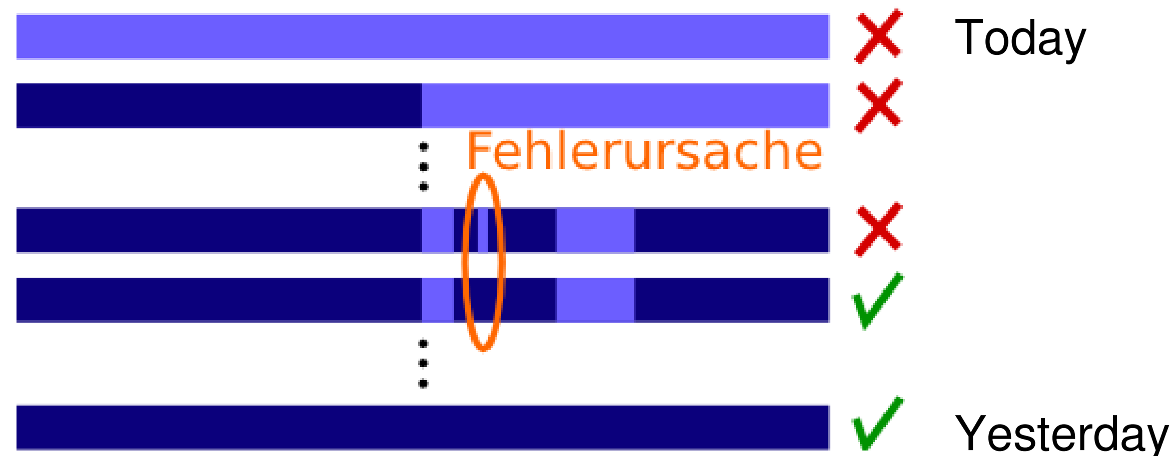
- *Konfiguration* ist eine Teilmenge aller Änderungen

  $c \subseteq \{\Delta_1, \Delta_2, \ldots, \Delta_n\}$
- Konfiguration hat in der Praxis meist zwei Eigenschaften:
  - Wenn eine Änderung einen Fehler erzeugt, dann erzeugt jede Konfiguration mit dieser Änderung den Fehler (Monotonie)
  - Ein Fehler wird von nur einer Änderungsmenge erzeugt, nicht von mehreren, unterschiedlichen (Eindeutigkeit)
- Außerdem nehmen wir zunächst Konsistenz an

Funktionsweise

Test erfolgreich

Test produziert erwarteten Fehler



Funktionsweise

Schritt	c_i	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	.	.	

- Teile die Konfiguration c in c_1 und c_2
- Test c_1 schlägt fehl – c_1 enthält fehlerbehaftete Änderung, fahre mit c_1 fort
- Test c_2 schlägt fehl – c_2 enthält fehlerbehaftete Änderung, fahre mit c_2 fort
- Beide Tests sind erfolgreich – Interferenz zwischen Änderungen aus c_1 und c_2

Eigenschaften

- Worst-Case Laufzeit ist linear
- Ist eine Konfiguration nicht monoton oder mehrdeutig, dann ergibt der Algorithmus
  - Für Interferenzen eine fehlerverursachende Änderungsmenge, die evtl. nicht minimal ist
  - Für Einzelfehler das richtige Ergebnis (Nicht-Monotonie) bzw. eine der fehlerverursachenden Änderungen (Mehrdeutigkeit)

Inkonsistenz

- Der bisherige Ansatz behandelt nur die Fälle ✓ und ✗
- Das wichtigste Problem bei der willkürlichen Anwendung von Änderungen sind inkonsistente Konfigurationen
- In einem solchen Fall kann kein Testergebnis ermittelt werden
  - Testlauf erzeugt unerwarteten Fehler ?

Inkonsistenz

- Es gibt zwei Konfigurationen, die immer konsistent sind

- Wahrscheinlichkeit für Konsistenz wird erhöht durch Anwendung von
  - Weniger Änderungen (Annäherung an „yesterday“)
  - Mehr Änderungen (Annäherung an „today“)
- Dazu: Teile Konfiguration c in kleinere Teile (2n statt 2)

Erweiterter Algorithmus

Teile c in n Teilmengen und betrachte die folgenden Fälle:

- **Fehler**
  - betrachte c_i, wenn test(c_i) = ✗
- **Interferenz**
  - betrachte c_i mit festem $\overline{c}_i$ und $\overline{c}_i$ mit festem c_i, wenn test(c_i) = test($\overline{c}_i$) = ✓
- **Präferenz**
  - betrachte c_i mit festem $\overline{c}_i$, wenn test(c_i) = ? und test($\overline{c}_i$) = ✓
- **Erneuter Versuch**
  - In allen anderen Fällen, wiederhole mit 2n Teilmengen

Erweiterter Algorithmus

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
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
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
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		

2, 3, 7 bilden Interferenz

Ausblick

Vermeidung von Inkonsistenzen

- Gruppierung von verwandten Änderungen bezüglich verschiedener Kriterien (chronologisch, räumlich, lexikalisch, syntaktisch, ...)
- Vorhersagen von Testergebnissen unter Ausnutzung zusätzlichen Wissens

Beispiel

- Kommandozeilenargumente sollen sortiert ausgegeben werden

```
~/ddsample$ ./sort_yesterday 14 3
Output: 3 14
~/ddsample$ ./sort_today 14 3
Output: 0 3
```

Beispiel

Yesterday

```
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");
```

Today

```
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");
```

Beispiel

```
 * 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
```

Beispiel

```
        a[j] = v;
      }
  } while (h != 1);

t main(int argc, char *argv[])

  int *a;
  int i;

  a = (int *)malloc((argc - 1) * sizeof(i
  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");
```

```
        a[var] = v;
      }
  } while (h != 1);
}

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

    a = (int *)malloc((argc) * sizeof
    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");
```

Beispiel

Yesterday

```
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)
```

Today

```
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;
      }
    }
  }
}
```

Beispiel

```
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'), (6, '\n22c\n     }\n.\n'), (7, '\n7,20c\n
   for (i = 0; i < size; ++i) {\n        for (j = 0; j < size - i - 1; ++j) {\n            if (
a[j] > a[j + 1]) {\n                int tmp = a[j];\n               a[j] = a[j + 1];\n
        a[j + 1] = tmp;\n            }\n.\n'), (8, '\n3a\n/* shell_sort: sortiert ein Array a d
er Laenge size aufsteigend */\n.'), (1, '37,38c\n    for (i = 1; i < outcount; i++)\n        pr
intf("%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 < argc; i++)\n
   a[i - 1] = atoi(argumente[i]);\n.\n'), (4, '\n28a\n    int outcount = argc;\n.\n'), (5, '\n2
5c\n/*\n * Die main-Methode.\n * Hier ist der Einsprungspunkt des Programmes.\n */\nint main(in
t argc, char **argumente)\n.\n'), (6, '\n22c\n     }\n.\n'), (7, '\n7,20c\n     for (i = 0; i <
 size; ++i) {\n        for (j = 0; j < size - i - 1; ++j) {\n            if (a[j] > a[j + 1]) {
\n                int tmp = a[j];\n                a[j] = a[j + 1];\n                a[j + 1] =
 tmp;\n            }\n.\n'), (8, '\n3a\n/* shell_sort: sortiert ein Array a der Laenge size auf
steigend */\n.')] fails
```

Beispiel

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


int *a;
int i;


a = (int *)malloc((argc - 1) * sizec
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);
```

```
/*
 * 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) * sizec
    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);
```

Beispiel

```
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')]
```

```
- 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--);
```

Zusammenfassung

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

Delta Debugging

- Kein Benutzereingriff nötig
- Kann mit komplizierten Details umgehen
- Integration in Regressionstests

Fragen?

Vielen Dank für die Aufmerksamkeit!

Literatur und Links

- Andreas Zeller: *Yesterday, my program worked. Today, it does not. Why?*, September 1999
- Andreas Zeller: *Why Programs Fail: A Guide to Systematic Debugging*, Dpunkt Verlag, 2005. ISBN 3-89864-279-8
- Delta Debugging - Software Engineering Chair (Prof. Zeller) - Saarland University http://www.st.cs.uni-sb.de/dd/