Delta debugging

Joanna Iwaniuk

7 czerwca 2011

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

Maurice Wilkes

Another effective technique is to explain your code to someone else. This will often cause you to explain the bug to yourself. Sometimes it takes no more than a few sentences, followed by an embarrassed "Never mind, I see what's wrong. Sorry to bother you." This works remarkably well; you can even use non-programmers as listeners. One university computer center kept a teddy bear near the help desk. Students with mysterious bugs were required to explain them to the bear before they could speak to a human counselor.

Brian Kernighan

Plan prezentacji

1. Co to jest delta debugging?
2. Do czego można go wykorzystać?
3. Implementacje delta debuggingu
4. Podsumowanie

Idea delta debuggingu

- http://www.st.cs.uni-saarland.de/dd/ - strona uniwersytetu w Saarbrücken
- idea algorytmu ddmin – znajdowanie minimalnego przypadku użycia, który powoduje błąd
- algorytm delta debuggingu

Oznaczenia, oznaczenia, oznaczenia...

- R – zbiór elementów budujących samolot (zbiór zmiennych czynników, które wpływają na działanie programu, np. dane wejściowe, kod źródłowy)
- $r_{✓}$ – dany z góry poprawnie działający samolot (wywołanie programu)
- $r_{✗}$ – dany z góry samolot, który spada zaraz po starcie
- Δ – zbiór wszystkich elementów, które odróżniają zły samolot od dobrego (fragmenty danych wejściowych, którymi różni się wywołanie $r_{✓}$ od $r_{✗}$ itp.)
- $\Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$ – dekompozycja Δ na mniejsze zbiory elementów (np. Δ_i – ekspres do kawy, jedna linia danych wejściowych)
- c – jakiś podzbiór elementów, które odróżniają jeden samolot od drugiego
- $c_{✓}$ – zbiór pusty
- $c_{✗} = \Delta$
- $test(c)$ – polega na wykonaniu próbnego lotu samolotem $r_{✓}$ z zaaplikowanymi zmianami ze zbioru c; wynik to ✓ jeśli samolot poleci poprawnie, ✗ jeśli samolot spadnie, ? wpp. (przetestowanie co się stanie po wykonaniu w działającym programie zmian c)
- algorytm dd zwraca $(c'_{✓}, c'_{✗})$, takie że $test(c'_{✗}) = ✗$ oraz $test(c'_{✓}) = ✓$ (im mniejsza moc $c'_{✗} \setminus c'_{✓}$ tym lepiej dla programisty)

Algorytm dd

$$dd(c_{\text{✘}}) = dd_2(\emptyset, c_{\text{✘}}, 2) \quad \text{where}$$

$$dd_2(c'_{\text{✔}}, c'_{\text{✘}}, n) = \begin{cases} dd_2(c'_{\text{✔}}, c'_{\text{✔}} \cup \Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✘} \text{ (“reduce to subset”)} \\ dd_2(c'_{\text{✘}} - \Delta_i, c'_{\text{✘}}, 2) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✔} \text{ (“increase to complement”)} \\ dd_2\big(c'_{\text{✔}} \cup \Delta_i, c'_{\text{✘}}, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✔} \text{ (“increase to subset”)} \\ dd_2\big(c'_{\text{✔}}, c'_{\text{✘}} - \Delta_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✘} \text{ (“reduce to complement”)} \\ dd_2\big(c'_{\text{✔}}, c'_{\text{✘}}, \min(2n, |\Delta|)\big) & \text{else if } n < |\Delta| \text{ (“increase granularity”)} \\ (c'_{\text{✔}}, c'_{\text{✘}}) & \text{otherwise (“done”)} \end{cases}$$

Źródło: Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input.

Algorytm dd – własności

- ozn. t – liczba wykonanych testów
- w pesymistycznym przypadku tak jak w ddmin: $t = |c_{\text{✗}}|^2 + 3 * |c_{\text{✗}}|$
- przypadek optymistyczny – wszystkie testy kończą się wynikiem ✔ lub ✗
- liczba testów w przypadku optymistycznym dwukrotnie mniejsza niż dla ddmin: $t \leq log_2(|c_{\text{✗}}|)$
- w takim przypadku $|dd(c_{\text{✗}})| = 1$

Algorytm dd – problemy

- program musi działać poprawnie przynajmniej w niektórych przypadkach
- zminimalizowany przypadek testowy może powodować inny błąd, niż ten powodowany przez pierwotny program
- istenieje rozwiązanie – algorytm ze zwiększoną precyzją
- czasem dane wejściowe są duże, a wykonanie pojedynczego testu długie

Algorytm dd – i co dalej?

Zaprezentowane podejście można zastosować równie dobrze do debugowania programu jak i szukania wady konstrukcji samolotu.

Przyjrzyjmy się zatem konkretnym zastosowaniom dd – czym mogą być $r_{\checkmark}$, $r_{\times}$, Δ?

Zastosowanie dd przy debugowaniu GCC

- $r_{✓}$– program GCC wywołany z pustym wejściem
- $r_{✗}$– GCC wywołany z wejściem, które powoduje jego błąd
- Δ_i – usunięcie pojedynczego znaku z danych wejściowych
- *test* – próba kompilacji pliku wejściowego, zwraca wynik ✓ jeśli kompilacja powiodła się, ✗ jeśli wystąpił błąd w działaniu kompilatora, ? wpp. (prawdopodobnie kompilacja nie powiodła się)
- dd znajdzie minimalne dane wejściowe, dla których działanie GCC kończy się błędem i maksymalne dane wejściowe, dla których błąd nie występuje

Program w C, który przy kompilacji powodował błąd GCC

```
#define SIZE 20

double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}

void copy(double to[], double from[], int count)
{
  int n = (count + 7) / 8;
  switch (count % 8) do {
    case 0: *to++ = *from++;
    case 7: *to++ = *from++;
    case 6: *to++ = *from++;
    case 5: *to++ = *from++;
    case 4: *to++ = *from++;
    case 3: *to++ = *from++;
    case 2: *to++ = *from++;
    case 1: *to++ = *from++;
  } while (--n > 0);
  return mult(to, 2);
}

int main(int argc, char *argv[])
{
  double x[SIZE], y[SIZE];
  double *px = x;

  while (px < x + SIZE)
    *px++ = (px - x) * (SIZE + 1.0);
  return copy(y, x, SIZE);
}
```

Źródło: Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input.

Wynik debugowania GCC (2)

Wynik działania algorytmu dd dla przykładu z GCC jest następujący:

(a) failing program

```
#define SIZE 20

double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }

  return z[n];
}
```

(b) passing program

```
#define SIZE 20

double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }

  return z[n];
}
```

Źródło: Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input.

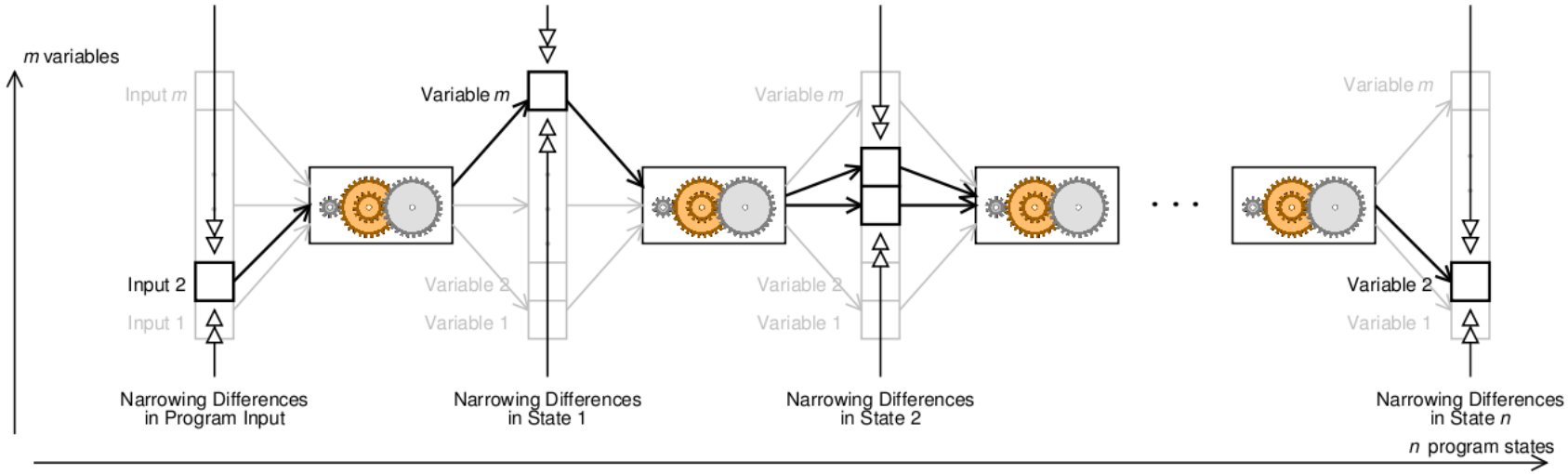
Zastosowanie dd (2) – izolacja stanów powodujących błąd

- $r_{\checkmark}$, $r_{\times}$– dwa wywołania programów podane przez użytkownika
- wykonanie programu traktujemy jako sekwencję stanów
- można traktować stan jako mapę zmiennych i ich wartości
- Δ_k – polega na zmianie wartości pojedynczej zmiennej
- algorytm dd odnajdzie maksymalny zbiór zmiennych, których wartość można podmienić, nie uzyskując błędu i minimalny zbiór zmiennych, których wartości nie można podmienić
- różnica między powyższymi to zmienne odpowiedzialne za wystąpienie błędu

Przeszukiwanie w czasie

- w kolejnych stanach za błąd mogą być odpowiedzialne różne zestawy zmiennych
- *cause transition* – występuje kiedy zmienna A, dotychczas odpowiedzialna za występowanie błędu, przestaje go powodować i można wyznaczyć inną zmienną odpowiedzialną za błąd
- *przeszukiwanie w czasie (ang. search in time)* polega na znajdowaniu miejsc z bezpośrednim *'cause transition'*
- algorytm nie wykonuje analizy kodu programu
- potrzebuje jedynie możliwości wykonania programu i zmieniania stanów programu w trakcie wykonania (np. przy użyciu programu GDB do debugowania programów w C)

Przeszukiwanie w czasie – przykład



Rysunek: Zmniejszanie przestrzeni zmiennych „odpowiedzialnych za błąd" w kolejnych stanach.

Źródło: Andreas Zeller. Isolating Cause-Effect Chains from Computer Programs.

Przykładowy kod w C do zdebugowania

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

return 0; }
```

Przeszukiwanie w czasie – przykład

Var	Value in $r_{✔}$	in $r_{✘}$
argc	**4**	**5**
argv[0]	"./sample"	"./sample"
argv[1]	**"9"**	**"11"**
argv[2]	**"8"**	**"14"**
argv[3]	**"7"**	**0x0** (NULL)
i'	1073834752	1073834752
j	1074077312	1074077312
h	1961	1961
size	**4**	**3**

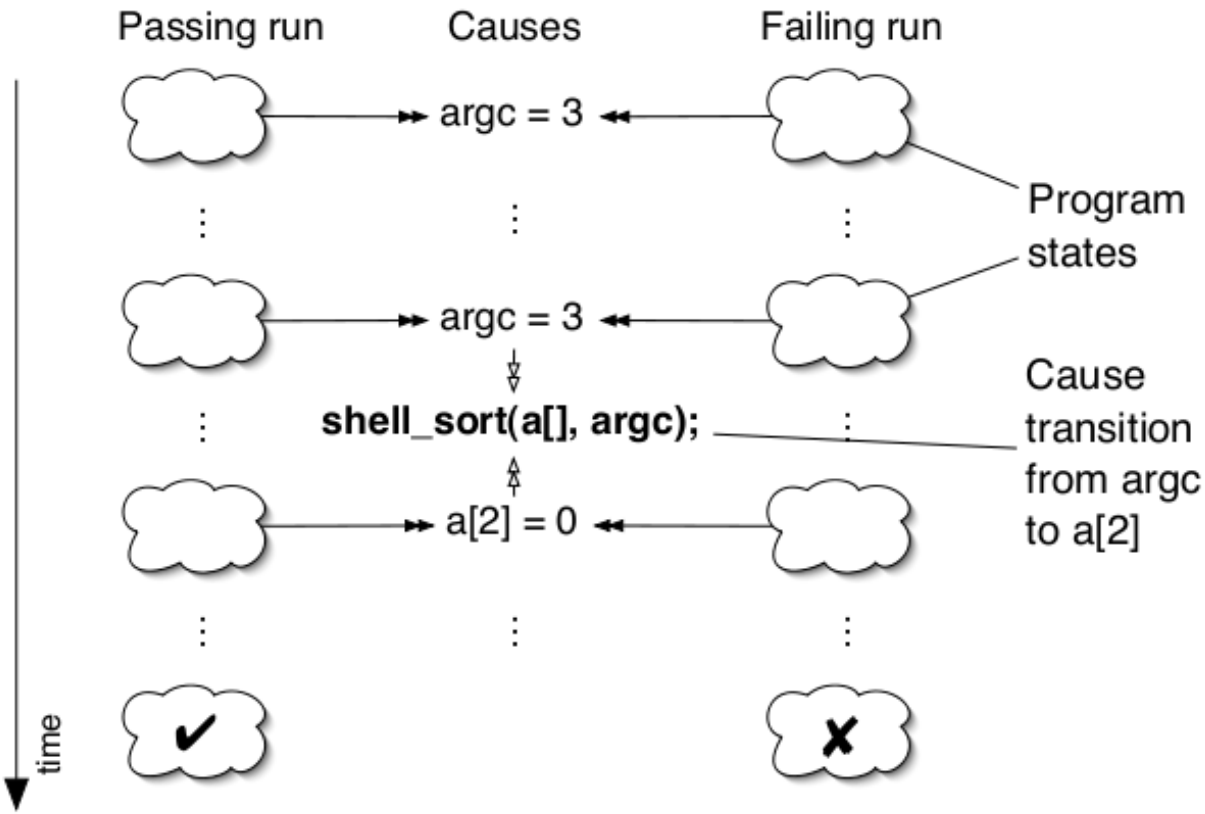
Var	Value in $r_{✔}$	in $r_{✘}$
i	**3**	**2**
a[0]	**9**	**11**
a[1]	**8**	**14**
a[2]	**7**	**0**
a[3]	1961	1961
a'[0]	**9**	**11**
a'[1]	**8**	**14**
a'[2]	**7**	**0**
a'[3]	1961	1961

Rysunek: Wartości zmiennych w danym stanie wykonania programu.

Źródło: Andreas Zeller. Isolating Cause-Effect Chains from Computer Programs.

Step	Line	Code	$r_{✓}$	$r_{✗}$	Vars	*cts* step
1	28	`int i=0;`	•	•	argc=3	2
⋮						
6	32	`for(i=0;i<argc-1;i++)`	•	•	argc=3	11
7	33	`a[i]=atoi(argv[i+1]);`	•	•		
8	32	`for(i=0;i<argc-1;i++)`	•	•	argc=3	12
9	33	`a[i]=atoi(argv[i+1]);`	•		?	
10	32	`for(i=0;i<argc-1;i++)`	•		?	
11	35	`shell_sort(a,argc);`	•	•	a[2]=0	3
⋮						
26	20	`if(i!=j)`	•	•	a[2]=0	4
27	21	`a[j]=v;`	•			
28	15	`for(i=h;i<size;i++)`	•	•	a[2]=0	9
29	17	`int v=a[i];`	•	•	a[2]=0	10
30	18	`for(j=i;j>=h&&a[...`	•	•	v=0	8
⋮						
35	20	`if(i!=j)`	•	•	v=0	5
36	21	`a[j]=v;`	•	•	v=0	7
37	15	`for(i=h;i<size;i++)`	•	•	a[0]=0	6
⋮						
44	37	`for(i=0;i<argc-1;i++)`	•	•	a[0]=0	1
45	38	`printf("%d ",a[i]);`	•	•		

Rysunek: Lokalizacja miejsc z *cause transition*.



Rysunek: Przyczynowo-skutkowy ciąg stanów znaleziony metodą przeszukiwania w czasie.

Źródło: Holger Cleve and Andreas Zeller. Locating Causes of Program Failures.

Rysunek: Wynik działania programu Igor przeszukującego w czasie.

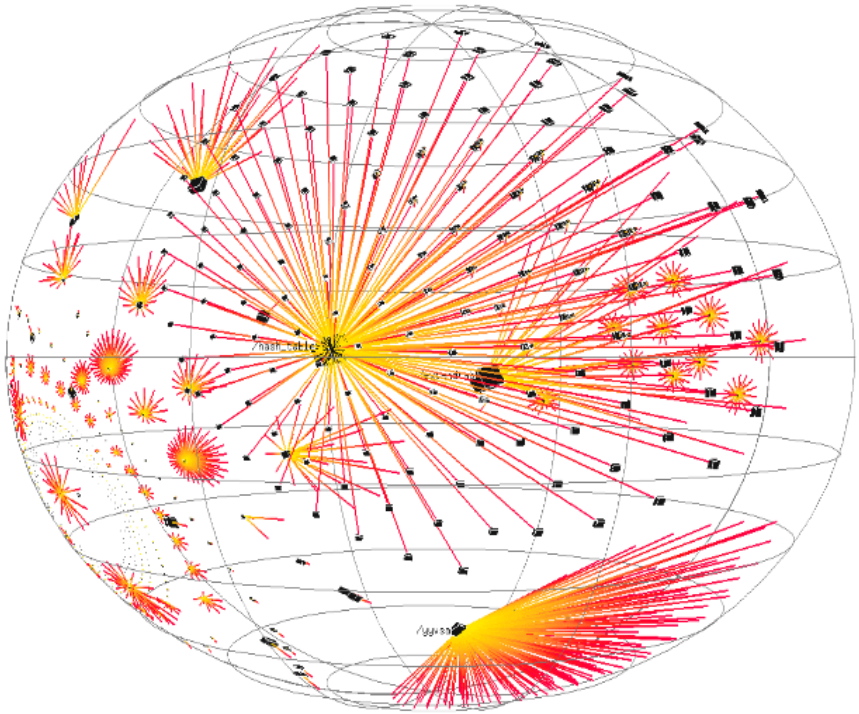
Przeszukiwanie w czasie – złożoność czasowa

- m – liczba *cause transitions*
- n – liczba kroków w progamie
- złożoność: $\mathcal{O}(mlogn) * delta\ debugging$
- przypomnienie: złożoność czasowa delta debuggingu = $\mathcal{O}(|c_x|^2)$
- taka złożoność wygląda całkiem nieźle, niestety użycie GDB jest dość kosztowne – czas potrzebny do zdebugowania gcc wyniósł 12h

Drobny problem przy izolacji zmiennych

- Δ_k polega na zmianie wartości pojedynczej zmiennej
- najłatwiej byłoby podmieniać kolejne wartości zmiennych w programu niedziałającego na te z programu działającego
- problem: zmiana niektórych wartości jest bezsensowna, np. wskaźników
- jak rozpoznać wartości, które można i warto zmienić?
- rozwiązanie: grafy pamięci (ang. memory graphs) – reprezentują zależności między zminnymi w programie

Memory graphs - przykład



Rysunek: Graf pamięci dla GCC.

Źródło: Andreas Zeller. Isolating Cause-Effect Chains from Computer Programs.

Inne zastosowania delta debuggingu

- Delta debugging dla programów wielowątkowych – wyznaczanie przeplotu, dla którego działanie programu daje błędny wynik
- Wykrywanie fragmentów kodu źródłowego, których wykonanie prowadzi do błędu
- Wykrywanie zdarzeń/interakcji komponentu, które mają znaczenie dla wystąpienia błędu.

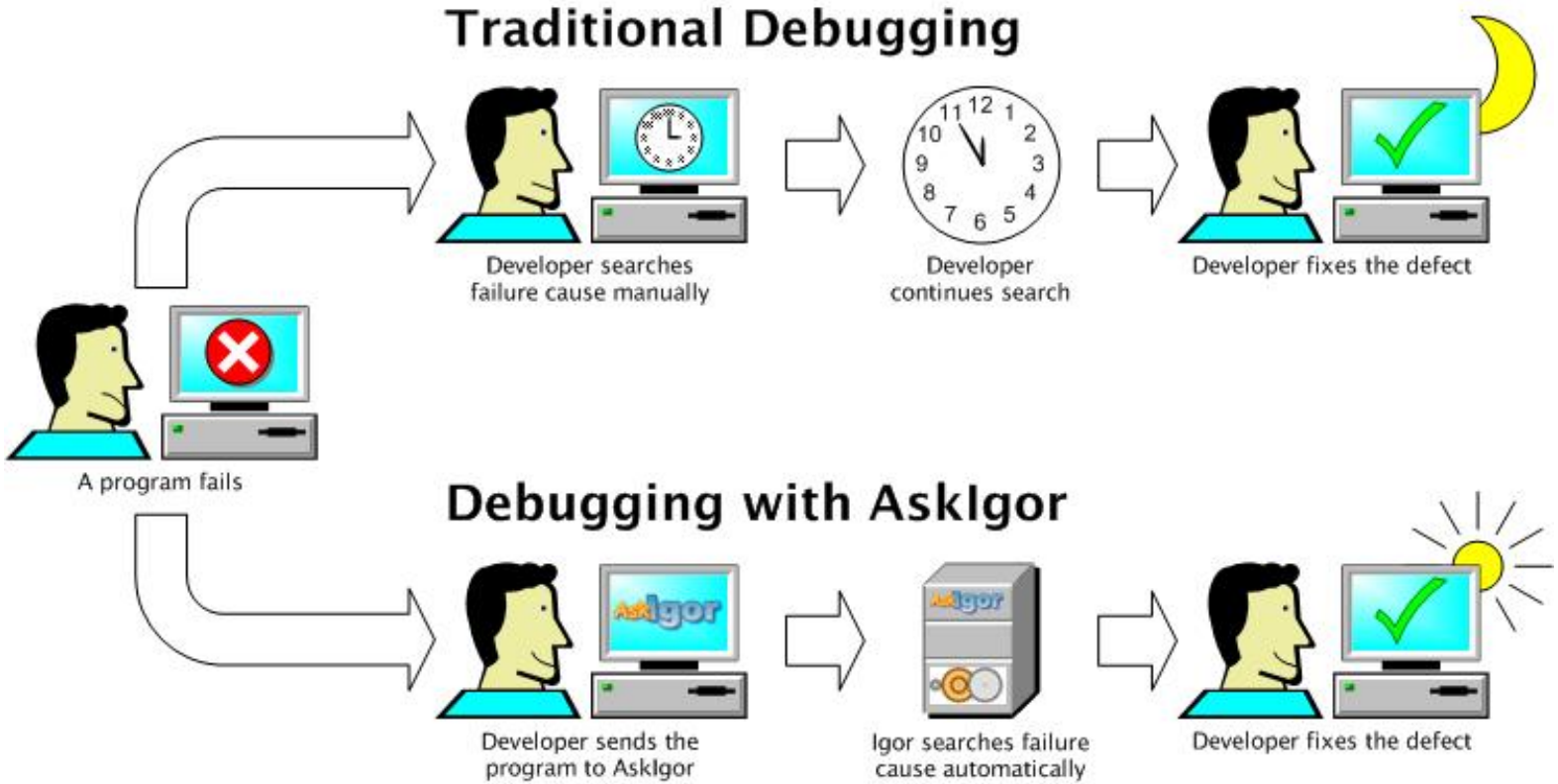
Implementacje dd

- Igor - izolacja stanów powodujących błąd dla programów w C
- wtyczki Eclipse o szerokiej gamie funkcjonalności: DDinput, DDchange, DDstate itd...

Implementacja dd - program Igor

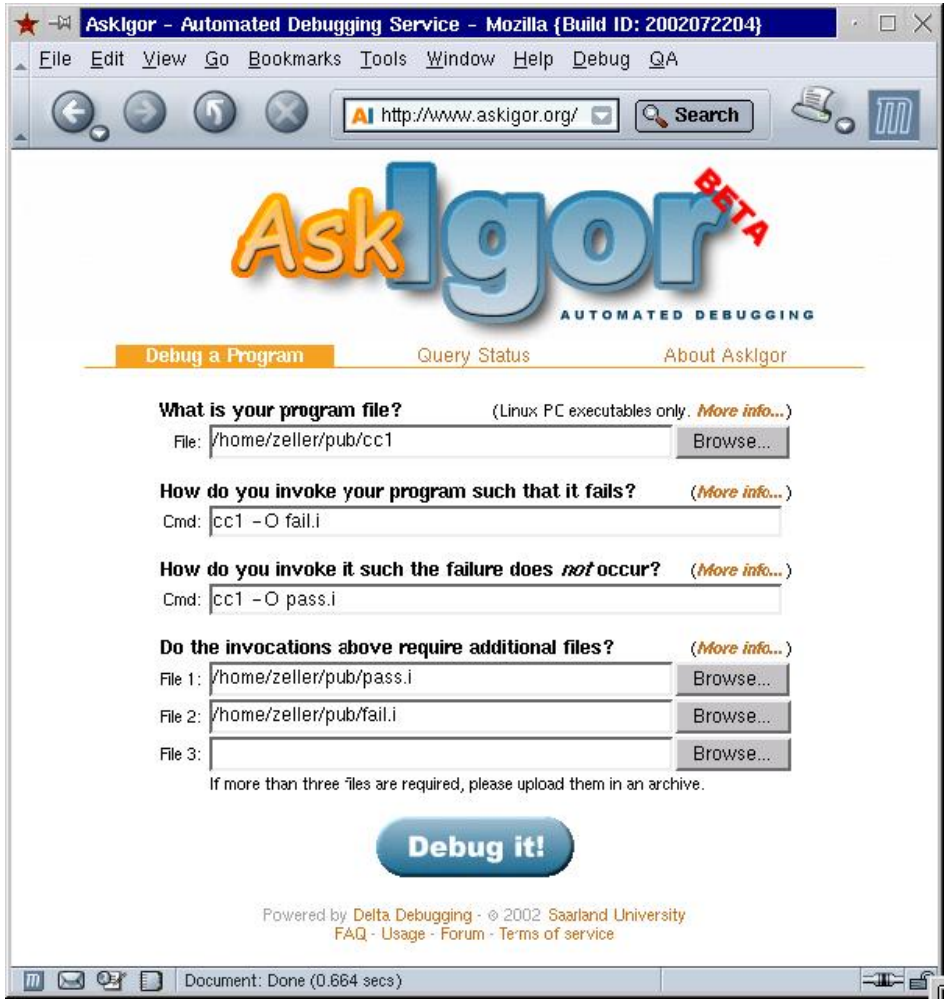
- służy do automatycznego debugowania programów w C – izolacja stanów powodujących błąd
- można go ściągnąć ze strony dd
- wersja, która powstała była eksperymentalna i obecnie nikt się nią już nie zajmuje :(

Działanie Igora według jego twórców



Źródło: http://www.st.cs.uni-saarland.de/askigor/images/

Strona internetowa AskIgor



Źródło: http://www.st.cs.uni-saarland.de/askigor/images/

AskIgor obecnie

Źródło: http://www.st.cs.uni-saarland.de/askigor/images/

Igor – prezentacja

Igor – ograniczenia

- debugowany program nie może wymagać interakcji z użytkownikiem
- ograniczone możliwości specyfikowania na czym polega poprawne i niepoprawne działanie programu
- nie działa dla programów, które są za duże
- „Igor will fail on a number of complex programs, and also on not-so-complex programs.” (ze stron pomocy man)

Implementacja dd – wtyczki Eclipse

- Aktualnie twórcy dd zajmują się integracją delta debuggingu ze środowiskiem Eclipse (podobno).
- Dotychczas utworzono wtyczki:
  - **DDinput** – izolacja fragmentów danych wejściowych, które są odpowiedzialne za wystąpienie błędu
  - **DDchange** – wykrywanie zmian w kodzie źródłowym, które są odpowiedzialne za błąd
  - **DDstate** – izolacja stanów powodujących błąd (tak jak w Igorze)
  - **eROSE** – podpowiada w jakich fragmentach kodu programista powinien wprowadzić zmiany

DDchange

- zintegrowany z narzędziem JUnit
- zapamiętuje poprzednie wersje kodu źródłowego, dla których testy JUnit dawały pozytywny wynik
- jeśli po jakichś zmianach w kodzie program przestaje „zdawać testy" to DDchange porównuje kod poprawny i niepoprawny i izoluje różnice, które wpłynęły na wyniki testów (delta debugging)

DDchange – prezentacja

DDchange – zalety

- W przeciwieństwie do wszystkich poprzednich zastosowań dd, DDchange pracuje na kodzie źródłowym.
- Można dokładnie określić jaki program jest poprawny, a jaki nie.
- Wygodny w użyciu.
- Podczas jego działania można zmieniać kod źródłowy – nie trzeba czekać na wynik debugowania.

DDchange – wady

- Często nie działa – problemy z zapamiętywaniem historii.
- Czasem mimo że działa to daje niezgodne z oczekiwaniami wyniki.
- Jeśli działa i daje wyniki zgodne z oczekiwaniami – jego wskazania bywają mało precyzyjne.

Możliwości dalszego rozwoju

- Połączenie delta debuggingu ze statycznymi metodami analizy poprawności.
- Integracja dd ze środowiskami programistycznymi.
- Subiektywna ocena: poprawienie istniejącego oprogramowania.

Podsumowanie

- Ideą delta debuggingu jest wyręczenie programisty w tym, czego bardzo nie lubi – znajdowaniu błędów we własnych programach.
- To brzmi świetnie! Niestety w obecnych implementacjach często nie działa zgodnie z oczekiwaniami.
- Wątpliwe czy w ogóle da się zaimplementować to tak, aby zawsze działało zgodnie z oczekiwaniami.
- W niczym jednak nie zaszkodzi spróbować tego użyć, skoro delta debuggery pracują w tle, a jedyną alternatywą są pluszowe misie ;)

Optymistyczne zakończenie

My ideal is, that no programmer should get home late, because he or she has been debugging. So my ideal is: at the end of the day press the button, look at your notes, go home. Come back the next morning, with a fresh mind, your computer is there to do the tedious stuff. It's not you, so let your computer do all the hard work.

Andreas Zeller w wywiadzie radiowym