

Debugging

Mohammad Mousavi

Halmstad University, Sweden

http://ceres.hh.se/mediawiki/DT8021_Ed_2015

Testing and Verification of Embedded Systems (DT8021),
April 20, 2015

Outline

Debugging

Sequential Slicing

Structured Slicing

Automated Debugging

Simplifying the Test-Case

Isolating the Causes

(Automated) Debugging: A Sorting Program

```
1: int main(int argc, char * argv[])
2: {
3:   int *a;
4:   int i;
5:   a = (int *) malloc( (argc - 1) * sizeof(int) );
6:   for (i = 0; i < argc - 1; i++)
7:     a[i] = atoi(argv[i + 1]);
8:   shell_sort(a, argc);
9:   printf(" Output: ");
10:  for (i = 0; i < argc - 1; i++)
11:    printf("%d ", a[i]);
12:  free(a);
13:  return 0;
14: }
```

```
1: void shell_sort(int a[], int size)
2: { int i, j; int h = 1;
3: do {
4:     h = h * 3 + 1;
5: } while (h <= size);
6: do {
7:     h /= 3;
8:     for (i = h; i < size; i++)
9:     {
10:         int v = a[i];
11:         for (j = i; j >= h && a[j - h] > v; j -= h)
12:             a[j] = a[j - h];
13:         if (i != j)    a[j] = v;
14:     }
15: } while (h != 1);
16: }
```

(Automated) Debugging: A Sorting Program

Once upon a time, a tester found the following bug:

```
$ ./simple 5 4 3 2 1 666666  
Output: 0 1 2 3 4 5
```

How do we find **the fault**?

Find and Focus

- ▶ Scientific method:
 1. assume,
 2. organize an experiment,
 3. if refuted, refine your assumption and repeat.possible formalization: invariants and assertions
- ▶ Observing: logging the value of infected variables
e.g., `print` command in `gdb`
- ▶ Watching: keeping an eye on infected variables
e.g., `break` and `watch` commands in `gdb`
- ▶ Slicing: find the slice responsible for infection
see the lecture on slicing



Getting Our Hands Dirty...

We use gdb (any other debugger will do)

- ▶ **Reproduce** the test:
run 5 4 3 2 1 666666 Damn, the tester was right!
(Not always that easy, try 55 4.)
- ▶ **Simplify** the test-case
run 5 4 3 2
- ▶ **Find** the possible the **origins**,
focus on a problem area,
e.g., `a[0]` and `shell_sort` (See **slicing** next...)
- ▶ **Isolate** the causes
what makes `a[0]` wrong?
compare it with the sane situation, what is different?
- ▶ **Correct** the problem



TRAFFIC

1. **T**rack the problem
2. **R**eproduce the failure
3. **A**utomate and simplify the test-case:
minimal test-case ⇐
4. **F**ind possible origins: where it first went wrong
5. **F**ocus on the most likely origins: what part of state is infected
6. **I**solate the chain: what causes the state to be infected ⇐
7. **C**orrect the defect



Outline

Debugging

Sequential Slicing

Structured Slicing

Automated Debugging

Simplifying the Test-Case

Isolating the Causes

DU-Paths vs. Slices

DU-path

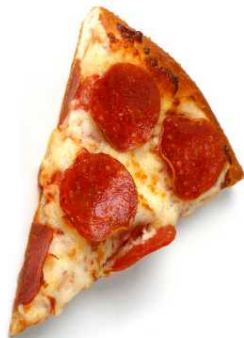
direct dependencies between **each** definition and use of a **single** variable.

Slice

An **executable** subset of the program

- ▶ capturing possible (**indirect**) **dependencies**
- ▶ among all definitions and uses
- ▶ influencing the value of a **set of variables**.

Also called: cone of influence reduction



Slicing: An Example

```
1: Input(x)
2: Input(y)
3: total := 0
4: sum := 0
5: if  $x \leq 1$  then
6:   sum := y
7: else
8:   Input(z)
9:   total :=  $x * y$ 
10: end if
11: Write(total, sum)
Slice on {total} at 11?
```

Slicing: An Example

```
1: Input(x)
2: Input(y)
3: total := 0
4: sum := 0
5: if  $x \leq 1$  then
6:   sum := y
7: else
8:   Input(z)
9:   total :=  $x * y$ 
10: endif
11: Write(total, sum)
Slice on {total} at 11?
```

Slicing: An Example

Slice on $\{total\}$ at 11:

```
1: Input(x)
2: Input(y)
3: total := 0
4: if  $x \leq 1$  then
5:
6: else
7:   total = x * y
8: end if
```

Slicing: An Example

- 1: Input(b)
- 2: $c := 1$
- 3: $d := 3$
- 4: $a := d$
- 5: $d := b + d$
- 6: $b := b + 1$
- 7: $a := b + c$
- 8: Write(a)

Slice on $\{d, c\}$ at 6?

Slicing: An Example

Slice on $\{d, c\}$ at 6:

- 1: Input(b)
- 2: $c := 1$
- 3: $d := 3$
- 4: $d := b + d$

$(6, \{d, c\})$ (in general (n, V)): **the slicing criterion**

Outline of the algorithm

Slice criterion (n, V)

- ▶ Statements in the slice: those **define** the **relevant** variables.
- ▶ At n , $v \in V$: relevant.
- ▶ A relevant $v \in DEF(m)$: v is **no more relevant** above m ,
- ▶ **but** then all variables in $USE(m)$ become relevant above m .

Relevant Variables

Given a slicing criterion (n, V) , $Relevant_0(m) =$

$$\begin{cases} V & \text{if } m = n + 1 \\ \{v \mid \exists_{m \rightarrow m'} (v \in relevant(m') \setminus DEF(m) \vee (DEF(m) \cap relevant(m') \neq \emptyset \wedge v \in USE(m)))\} & \text{otherwise} \end{cases}$$

Relevant Variables

Given a slicing criterion (n, V) , $Relevant_0(m) =$

$$\begin{cases} 1) V & \text{if } m = n + 1 \\ 2a) \{v \mid \exists_{m \rightarrow m'} (v \in relevant(m') \setminus DEF(m)) \vee \\ 2b) \quad (DEF(m) \cap relevant(m') \neq \emptyset \wedge v \in USE(m))\} & \text{otherwise} \end{cases}$$

- 1) base case: all variables in V are initially relevant
- 2a) v remains relevant: has been relevant below and not defined at m
- 2b) v becomes relevant: defines relevant variables

Slicing: An Example

Slicing criterion: $(6, \{d, c\})$?

$Relevant_0(m) =$

$$\begin{cases} 1) V & \text{if } m = n + 1 \\ 2a) \{v \mid \exists_{m \rightarrow m'} (v \in relevant(m') \setminus DEF(m) \vee \\ 2b) \quad (DEF(m) \cap relevant(m') \neq \emptyset \wedge v \in USE(m))\} & \text{otherwise} \end{cases}$$

m **Relevant₀(m)**

1 Input(b)

2 $c := 1$

3 $d := 3$

4 $a := d$

5 $d := b + d$

6 $b := b + 1$

$\{d, c\}$

Slicing: An Example

Slicing criterion: $(6, \{d, c\})$?

$Relevant_0(m) =$

$$\begin{cases} 1) V & \text{if } m = n + 1 \\ 2a) \{v \mid \exists_{m \rightarrow m'} (v \in relevant(m') \setminus DEF(m)) \vee \\ 2b) (DEF(m) \cap relevant(m') \neq \emptyset \wedge v \in USE(m))\} & \text{otherwise} \end{cases}$$

m **Relevant₀(m)**

1 Input(b)

2 $c := 1$

3 $d := 3$

4 $a := d$

5 $d := b + d$

6 $b := b + 1$ $\{d, c\}$

$\{d, c\}$

Slicing: An Example

Slicing criterion: $(6, \{d, c\})$?

$Relevant_0(m) =$

$$\begin{cases} 1) V & \text{if } m = n + 1 \\ 2a) \{v \mid \exists_{m \rightarrow m'} (v \in relevant(m') \setminus DEF(m)) \vee \\ 2b) (DEF(m) \cap relevant(m') \neq \emptyset \wedge v \in USE(m))\} & \text{otherwise} \end{cases}$$

m **Relevant₀(m)**

1 Input(b)

2 $c := 1$

3 $d := 3$

4 $a := d$

5 $d := b + d$ $\{c, b, d\}$

6 $b := b + 1$ $\{d, c\}$

$\{d, c\}$

Slicing: An Example

Slicing criterion: $(6, \{d, c\})$?

$Relevant_0(m) =$

$$\begin{cases} 1) V & \text{if } m = n + 1 \\ 2a) \{v \mid \exists_{m \rightarrow m'} (v \in relevant(m') \setminus DEF(m)) \vee \\ 2b) (DEF(m) \cap relevant(m') \neq \emptyset \wedge v \in USE(m))\} & \text{otherwise} \end{cases}$$

m **Relevant₀(m)**

1 Input(b)

2 $c := 1$

3 $d := 3$

4 $a := d$ $\{c, b, d\}$

5 $d := b + d$ $\{c, b, d\}$

6 $b := b + 1$ $\{d, c\}$

$\{d, c\}$

Slicing: An Example

Slicing criterion: $(6, \{d, c\})$?

$Relevant_0(m) =$

$$\begin{cases} 1) V & \text{if } m = n + 1 \\ 2a) \{v \mid \exists_{m \rightarrow m'} (v \in relevant(m') \setminus DEF(m)) \vee \\ 2b) (DEF(m) \cap relevant(m') \neq \emptyset \wedge v \in USE(m))\} & \text{otherwise} \end{cases}$$

m **Relevant₀(m)**

1 Input(b)

2 $c := 1$

3 $d := 3$ $\{c, b\}$

4 $a := d$ $\{c, b, d\}$

5 $d := b + d$ $\{c, b, d\}$

6 $b := b + 1$ $\{d, c\}$

$\{d, c\}$

Slicing: An Example

Slicing criterion: $(6, \{d, c\})$?

$Relevant_0(m) =$

$$\begin{cases} 1) V & \text{if } m = n + 1 \\ 2a) \{v \mid \exists_{m \rightarrow m'} (v \in relevant(m') \setminus DEF(m)) \} & \text{otherwise} \\ 2b) (DEF(m) \cap relevant(m') \neq \emptyset \wedge v \in USE(m)) \} \end{cases}$$

m **Relevant₀(m)**

1 Input(b)

2 $c := 1$ $\{b\}$

3 $d := 3$ $\{c, b\}$

4 $a := d$ $\{c, b, d\}$

5 $d := b + d$ $\{c, b, d\}$

6 $b := b + 1$ $\{d, c\}$

$\{d, c\}$

Slicing: An Example

Slicing criterion: $(6, \{d, c\})$?

$Relevant_0(m) =$

$$\begin{cases} 1) V & \text{if } m = n + 1 \\ 2a) \{v \mid \exists_{m \rightarrow m'} (v \in relevant(m') \setminus DEF(m)) \} & \text{otherwise} \\ 2b) (DEF(m) \cap relevant(m') \neq \emptyset \wedge v \in USE(m)) \end{cases}$$

m **Relevant₀(m)**

1 Input(b)	\emptyset
2 $c := 1$	$\{b\}$
3 $d := 3$	$\{c, b\}$
4 $a := d$	$\{c, b, d\}$
5 $d := b + d$	$\{c, b, d\}$
6 $b := b + 1$	$\{d, c\}$
	$\{d, c\}$

Slicing Sequential Programs

$m \in \text{Slice}_0(n, V)$ when

1. $n = m$ and $\text{DEF}(m) \cap V \neq \emptyset$, or
2. $m \rightarrow \dots \rightarrow n$ and
there exists an m' such that $m \rightarrow m'$ and
 $\text{DEF}(m) \cap \text{Relevant}_0(m') \neq \emptyset$

Slicing Sequential Programs

$m \in \text{Slice}_0(n, V)$ when

1. $n = m$ and $\text{DEF}(m) \cap V \neq \emptyset$, or
2. $m \rightarrow \dots \rightarrow n$ and
there exists an m' such that $m \rightarrow m'$ and
 $\text{DEF}(m) \cap \text{Relevant}_0(m') \neq \emptyset$

m	Relevant₀(m)	DEF(m)	∈ Slice₀(6, {d, c})
1 Input(b)	\emptyset		
2 $c := 1$	$\{b\}$		
3 $d := 3$	$\{c, b\}$		
4 $a := d$	$\{c, b, d\}$		
5 $d := b + d$	$\{c, b, d\}$		
6 $b := b + 1$	$\{d, c\}$		
	$\{d, c\}$		

Slicing Sequential Programs

$m \in \text{Slice}_0(n, V)$ when

1. $n = m$ and $\text{DEF}(m) \cap V \neq \emptyset$, or
2. $m \rightarrow \dots \rightarrow n$ and
there exists an m' such that $m \rightarrow m'$ and
 $\text{DEF}(m) \cap \text{Relevant}_0(m') \neq \emptyset$

m	Relevant₀(m)	DEF(m)	∈ Slice₀(6, {d, c})
1 Input(b)	\emptyset		
2 $c := 1$	$\{b\}$		
3 $d := 3$	$\{c, b\}$		
4 $a := d$	$\{c, b, d\}$		
5 $d := b + d$	$\{c, b, d\}$		
6 $b := b + 1$	$\{d, c\}$ $\{d, c\}$	$\{b\}$	×

Slicing Sequential Programs

$m \in \text{Slice}_0(n, V)$ when

1. $n = m$ and $\text{DEF}(m) \cap V \neq \emptyset$, or
2. $m \rightarrow \dots \rightarrow n$ and
there exists an m' such that $m \rightarrow m'$ and
 $\text{DEF}(m) \cap \text{Relevant}_0(m') \neq \emptyset$

m	Relevant₀(m)	DEF(m)	∈ Slice₀(6, {d, c})
1 Input(b)	\emptyset		
2 $c := 1$	$\{b\}$		
3 $d := 3$	$\{c, b\}$		
4 $a := d$	$\{c, b, d\}$		
5 $d := b + d$	$\{c, b, d\}$	$\{d\}$	✓
6 $b := b + 1$	$\{d, c\}$ $\{d, c\}$	$\{b\}$	×

Slicing Sequential Programs

$m \in \text{Slice}_0(n, V)$ when

1. $n = m$ and $\text{DEF}(m) \cap V \neq \emptyset$, or
2. $m \rightarrow \dots \rightarrow n$ and
there exists an m' such that $m \rightarrow m'$ and
 $\text{DEF}(m) \cap \text{Relevant}_0(m') \neq \emptyset$

m	Relevant₀(m)	DEF(m)	∈ Slice₀(6, {d, c})
1 Input(b)	\emptyset		
2 $c := 1$	$\{b\}$		
3 $d := 3$	$\{c, b\}$		
4 $a := d$	$\{c, b, d\}$	$\{a\}$	×
5 $d := b + d$	$\{c, b, d\}$	$\{d\}$	✓
6 $b := b + 1$	$\{d, c\}$ $\{d, c\}$	$\{b\}$	×

Slicing Sequential Programs

$m \in \text{Slice}_0(n, V)$ when

1. $n = m$ and $\text{DEF}(m) \cap V \neq \emptyset$, or
2. $m \rightarrow \dots \rightarrow n$ and
there exists an m' such that $m \rightarrow m'$ and
 $\text{DEF}(m) \cap \text{Relevant}_0(m') \neq \emptyset$

m	$\text{Relevant}_0(m)$	$\text{DEF}(m)$	$\in \text{Slice}_0(6, \{d, c\})$
1 Input(b)	\emptyset		
2 $c := 1$	$\{b\}$		
3 $d := 3$	$\{c, b\}$	$\{d\}$	✓
4 $a := d$	$\{c, b, d\}$	$\{a\}$	×
5 $d := b + d$	$\{c, b, d\}$	$\{d\}$	✓
6 $b := b + 1$	$\{d, c\}$ $\{d, c\}$	$\{b\}$	×

Slicing Sequential Programs

$m \in \text{Slice}_0(n, V)$ when

1. $n = m$ and $\text{DEF}(m) \cap V \neq \emptyset$, or
2. $m \rightarrow \dots \rightarrow n$ and
there exists an m' such that $m \rightarrow m'$ and
 $\text{DEF}(m) \cap \text{Relevant}_0(m') \neq \emptyset$

m	Relevant₀(m)	DEF(m)	∈ Slice₀(6, {d, c})
1 Input(b)	\emptyset		
2 $c := 1$	$\{b\}$	$\{c\}$	✓
3 $d := 3$	$\{c, b\}$	$\{d\}$	✓
4 $a := d$	$\{c, b, d\}$	$\{a\}$	×
5 $d := b + d$	$\{c, b, d\}$	$\{d\}$	✓
6 $b := b + 1$	$\{d, c\}$ $\{d, c\}$	$\{b\}$	×

Slicing Sequential Programs

$m \in \text{Slice}_0(n, V)$ when

1. $n = m$ and $\text{DEF}(m) \cap V \neq \emptyset$, or
2. $m \rightarrow \dots \rightarrow n$ and
there exists an m' such that $m \rightarrow m'$ and
 $\text{DEF}(m) \cap \text{Relevant}_0(m') \neq \emptyset$

m	Relevant₀(m)	DEF(m)	∈ Slice₀(6, {d, c})
1 Input(b)	\emptyset	$\{b\}$	✓
2 $c := 1$	$\{b\}$	$\{c\}$	✓
3 $d := 3$	$\{c, b\}$	$\{d\}$	✓
4 $a := d$	$\{c, b, d\}$	$\{a\}$	×
5 $d := b + d$	$\{c, b, d\}$	$\{d\}$	✓
6 $b := b + 1$	$\{d, c\}$ $\{d, c\}$	$\{b\}$	×

Outline

Debugging

Sequential Slicing

Structured Slicing

Automated Debugging

Simplifying the Test-Case

Isolating the Causes

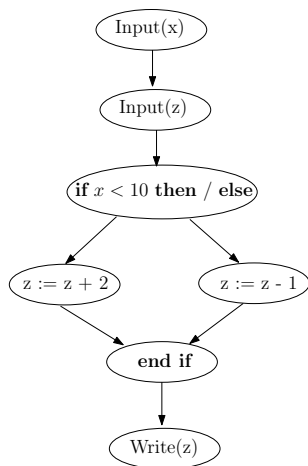
Slicing Programs with Conditions

```

1: Input(x)
2: Input(z)
3: if  $x < 10$  then
4:    $z := z + 2$ ;
5: else
6:    $z := z - 1$ ;
7: end if
8: Write(z)

```

Slice wrt. the criterion $(3, \{x\})$?



Slicing Programs with Conditions

Slice wrt. the criterion $(3, \{x\})$?

m	Relevant₀(m)	DEF(m)	∈ Slice₀(3, {x})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
		{x}	

Slicing Programs with Conditions

Slice wrt. the criterion $(3, \{x\})$?

m	Relevant₀(m)	DEF(m)	∈ Slice₀(3, {x})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else	$\{x\}$ $\{x\}$	\emptyset	×

Slicing Programs with Conditions

Slice wrt. the criterion $(3, \{x\})$?

m	Relevant₀(m)	DEF(m)	∈ Slice₀(3, {x})
1 Input(x)			
2 Input(z)	{x}	{z}	×
3,5 if $x < 10$ then / else	{x}	∅	×
	{x}		

Slicing Programs with Conditions

Slice wrt. the criterion $(3, \{x\})$?

m	Relevant₀(m)	DEF(m)	∈ Slice₀(3, {x})
1 Input(x)	\emptyset	$\{x\}$	✓
2 Input(z)	$\{x\}$	$\{z\}$	×
3,5 if $x < 10$ then / else	$\{x\}$ $\{x\}$	\emptyset	×

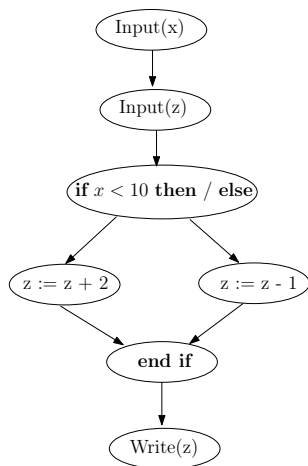
Slicing Programs with Conditions

```

1: Input(x)
2: Input(z)
3: if  $x < 10$  then
4:    $z := z + 2$ ;
5: else
6:    $z := z - 1$ ;
7: end if
8: Write(z)

```

Slice wrt. the criterion $(8, \{z\})$?



Slicing Programs with Conditions

m	Relevant₀(m)	DEF(m)	∈ Slice₀(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
4 $z := z + 2$			
6 $z := z - 1$			
7 end if			
8 Write(z)			
		{z}	

Slicing Programs with Conditions

m	Relevant₀(m)	DEF(m)	∈ Slice₀(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
4 $z := z + 2$			
6 $z := z - 1$			
7 end if			
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

m	Relevant₀(m)	DEF(m)	∈ Slice₀(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
4 $z := z + 2$			
6 $z := z - 1$			
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

m	Relevant₀(m)	DEF(m)	∈ Slice₀(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
4 $z := z + 2$			
6 $z := z - 1$	{z}	{z}	✓
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

m	Relevant₀(m)	DEF(m)	∈ Slice₀(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
4 $z := z + 2$	{z}	{z}	✓
6 $z := z - 1$	{z}	{z}	✓
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

m	Relevant₀(m)	DEF(m)	∈ Slice₀(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else	{z}	∅	×
4 $z := z + 2$	{z}	{z}	✓
6 $z := z - 1$	{z}	{z}	✓
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

m	Relevant₀(m)	DEF(m)	∈ Slice₀(8, {z})
1 Input(x)			
2 Input(z)	∅	{z}	✓
3,5 if $x < 10$ then / else	{z}	∅	×
4 $z := z + 2$	{z}	{z}	✓
6 $z := z - 1$	{z}	{z}	✓
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

m	Relevant₀(m)	DEF(m)	∈ Slice₀(8, {z})
1 Input(x)	∅	{x}	×
2 Input(z)	∅	{z}	✓
3,5 if $x < 10$ then / else	{z}	∅	×
4 $z := z + 2$	{z}	{z}	✓
6 $z := z - 1$	{z}	{z}	✓
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Slicing Structured Programs: Informal Idea

1. Start with sequential slicing algorithm: $Slice_0(n, v)$
2. Find all conditionals $Cond_{k+1}(n, V)$ influencing $m \in Slice_k(n, v)$
3. Add the following node to $Slice_k(n, V)$, the result: $Slice_{k+1}(n, V)$
 - 3.1 the conditional in $c \in Cond_k n, V$ and
 - 3.2 those statement influencing the conditions of c
4. repeat 2 until a fixed-point

(Inverse) Denominators

$m \in IDen(n)$ (m inversely denominates n)
when m appears in **all** paths $n \rightarrow \dots \rightarrow n_t$.

$m = NIDen(n)$ (the nearest inverse denominator of n) when
 $m \in IDen(n)$ and
for all $m' \in IDen(n)$ either $m = m'$ or there is a simple path
 $m \rightarrow \dots \rightarrow m'$.

$m \in Infl(n)$ (m is influenced by n) when
 m appears in a path from n to $NIDen(n)$
($m \neq n$, $m \neq NIDen(n)$, $NIDen(n)$ may not appear in the path).

Slicing Programs with Conditions

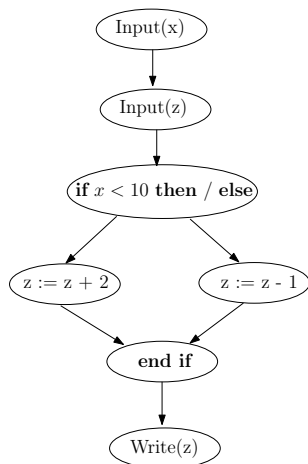
```

1: Input(x)
2: Input(z)
3: if  $x < 10$  then
4:    $z = z + 2$ ;
5: else
6:    $z = z - 1$ ;
7: end if
8: Write(z)
  
```

NIDen(1)? *Infl*(1)?

NIDen(2)? *Infl*(2)?

NIDen(3)? *Infl*(3)?



Slicing Programs with Conditions

```

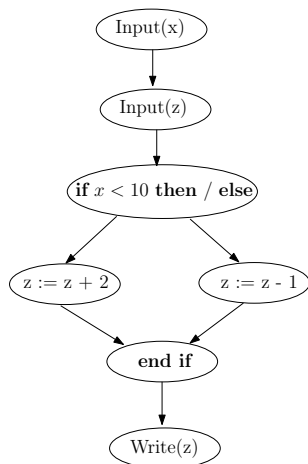
1: Input(x)
2: Input(z)
3: if  $x < 10$  then
4:    $z = z + 2$ ;
5: else
6:    $z = z - 1$ ;
7: end if
8: Write(z)

```

NIDen(1)? 2. *Infl*(1)? \emptyset .

NIDen(2)? *Infl*(2)?

NIDen(3)? *Infl*(3)?



Slicing Programs with Conditions

```

1: Input(x)
2: Input(z)
3: if  $x < 10$  then
4:    $z = z + 2$ ;
5: else
6:    $z = z - 1$ ;
7: end if
8: Write(z)

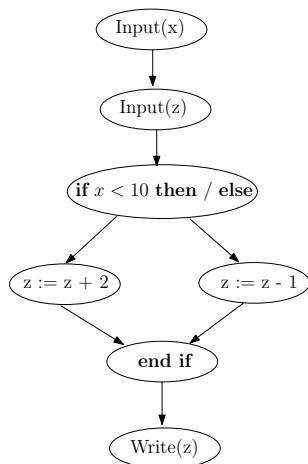
```

$NIDen(1)?$ 2. $Infl(1)?$ \emptyset .

$NIDen(2)?$ 3. $Infl(2)?$ \emptyset .

Observation, for **sequential** nodes $Infl(n) = \emptyset$.

$NIDen(3)?$ $Infl(3)?$



Slicing Programs with Conditions

```

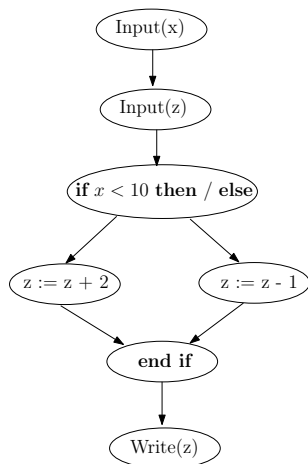
1: Input(x)
2: Input(z)
3: if  $x < 10$  then
4:    $z = z + 2$ ;
5: else
6:    $z = z - 1$ ;
7: end if
8: Write(z)
  
```

$NIDen(1)?$ 2. $Infl(1)?$ \emptyset .

$NIDen(2)?$ 3. $Infl(2)?$ \emptyset .

Observation, for **sequential** nodes $Infl(n) = \emptyset$.

$NIDen(3)?$ 7. $Infl(3)?$ $\{4, 6\}$.



Slicing Structured Programs

Given a slicing criterion (n, V) :

$m \in \text{Cond}_{k+1}(n, V)$ (conditions influencing $\text{Slice}_k(n, V)$) when there exists $m' \in \text{Slice}_k(n, V)$ and $m' \in \text{Infl}(m)$.

Slicing Structured Programs

Given a slicing criterion (n, V) :

$m \in \mathit{Cond}_{k+1}(n, V)$ (conditions influencing $\mathit{Slice}_k(n, V)$) when there exists $m' \in \mathit{Slice}_k(n, V)$ and $m' \in \mathit{Infl}(m)$.

$v \in \mathit{Relevant}_{k+1}(m)$ when

$v \in \mathit{Relevant}_k(m)$ or

there exists an $m' \in \mathit{Cond}_{k+1}(n, V)$ and

$v \in \mathit{Relevant}_0(m)$ w.r.t. the slicing criterion $(m', \mathit{USE}(m'))$.

Slicing Structured Programs

Given a slicing criterion (n, V) :

$m \in \mathit{Cond}_{k+1}(n, V)$ (conditions influencing $\mathit{Slice}_k(n, V)$) when there exists $m' \in \mathit{Slice}_k(n, V)$ and $m' \in \mathit{Infl}(m)$.

$v \in \mathit{Relevant}_{k+1}(m)$ when

$v \in \mathit{Relevant}_k(m)$ or

there exists an $m' \in \mathit{Cond}_{k+1}(n, V)$ and

$v \in \mathit{Relevant}_0(m')$ w.r.t. the slicing criterion $(m', \mathit{USE}(m'))$.

$m \in \mathit{Slice}_{k+1}(n, V)$ when

$m \in \mathit{Cond}_{k+1}(n, V)$ or

there exists an m' such that $m \rightarrow m'$ and

$\mathit{DEF}(m) \cap \mathit{Relevant}_{k+1}(m') \neq \emptyset$.

Slicing Programs with Conditions

Slice wrt. $(8, \{z\})$

- 1: Input(x)
- 2: Input(z)
- 3: **if** $x < 10$ **then**
- 4: $z = z + 2$;
- 5: **else**
- 6: $z = z - 1$;
- 7: **end if**
- 8: Write(z)

Slicing Programs with Conditions

Slice wrt. $(8, \{z\})$

- 1: Input(x)
- 2: Input(z)
- 3: **if** $x < 10$ **then**
- 4: $z = z + 2$;
- 5: **else**
- 6: $z = z - 1$;
- 7: **end if**
- 8: Write(z)

$Slice_0(8, \{z\}) = \{2, 4, 6\}$.

Slicing Programs with Conditions

Slice wrt. $(8, \{z\})$

- 1: Input(x)
- 2: Input(z)
- 3: **if** $x < 10$ **then**
- 4: $z = z + 2$;
- 5: **else**
- 6: $z = z - 1$;
- 7: **end if**
- 8: Write(z)

$Slice_0(8, \{z\}) = \{2, 4, 6\}$.

$m \in Cond_{k+1}(n, V)$ (conditions influencing $Slice_k(n, V)$) when there exists $m' \in Slice_k(n, V)$ and $m' \in Infl(m)$.

Slicing Programs with Conditions

Slice wrt. $(8, \{z\})$

- 1: Input(x)
- 2: Input(z)
- 3: **if** $x < 10$ **then**
- 4: $z = z + 2$;
- 5: **else**
- 6: $z = z - 1$;
- 7: **end if**
- 8: Write(z)

$Slice_0(8, \{z\}) = \{2, 4, 6\}$.

$Cond_1(8, \{z\}) = \{3\}$

$Slice_1(8, \{z\})?$

Slicing Programs with Conditions

m	Relevant₁(m)	DEF(m)	∈ Slice₁(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
4 $z := z + 2$			
6 $z := z - 1$			
7 end if			
8 Write(z)			
		{z}	

Slicing Programs with Conditions

m	Relevant₁(m)	DEF(m)	∈ Slice₁(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
4 $z := z + 2$			
6 $z := z - 1$			
7 end if			
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

m	Relevant₁(m)	DEF(m)	∈ Slice₁(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
4 $z := z + 2$			
6 $z := z - 1$			
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

m	Relevant₁(m)	DEF(m)	∈ Slice₁(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
4 $z := z + 2$			
6 $z := z - 1$	{z}	{z}	✓
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

m	Relevant₁(m)	DEF(m)	∈ Slice₁(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else			
4 $z := z + 2$	{z}	{z}	✓
6 $z := z - 1$	{z}	{z}	✓
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

m	Relevant₁(m)	DEF(m)	∈ Slice₁(8, {z})
1 Input(x)			
2 Input(z)			
3,5 if $x < 10$ then / else	$\{z, x\}$	\emptyset	×
4 $z := z + 2$	$\{z\}$	$\{z\}$	✓
6 $z := z - 1$	$\{z\}$	$\{z\}$	✓
7 end if	$\{z\}$	\emptyset	×
8 Write(z)	$\{z\}$	\emptyset	×
	$\{z\}$		

Slicing Programs with Conditions

m	Relevant₁(m)	DEF(m)	∈ Slice₁(8, {z})
1 Input(x)			
2 Input(z)	{x}	{z}	✓
3,5 if $x < 10$ then / else	{z, x}	∅	×
4 $z := z + 2$	{z}	{z}	✓
6 $z := z - 1$	{z}	{z}	✓
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Slicing Programs with Conditions

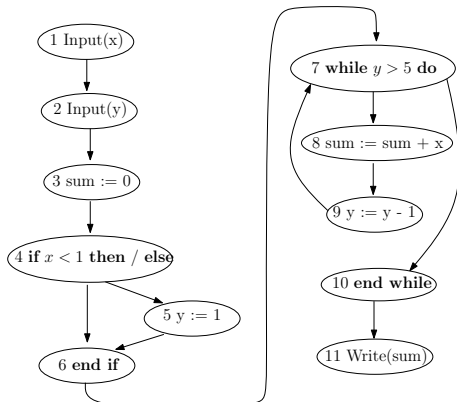
m	Relevant₁(m)	DEF(m)	∈ Slice₁(8, {z})
1 Input(x)	∅	{x}	✓
2 Input(z)	{x}	{z}	✓
3,5 if $x < 10$ then / else	{z, x}	∅	×
4 $z := z + 2$	{z}	{z}	✓
6 $z := z - 1$	{z}	{z}	✓
7 end if	{z}	∅	×
8 Write(z)	{z}	∅	×
	{z}		

Another Example

Slice wrt. $(11, \{sum\})$?

```

1: Input(x)
2: Input(y)
3: sum := 0
4: if  $x < 1$  then
5:    $y := 1$ 
6: end if
7: while  $y \geq 1$  do
8:    $sum := sum + x$ 
9:    $y := y - 1$ 
10: end while
11: Write(sum)
  
```



m	DEF(m)	Relevant ₀ (m)	Slice ₀	Cond ₁	Rel ₁	Slice ₁
1	{x}	∅	√,	×	∅	√
2	{y}	{x}	×	×	{x}	√
3	{sum}	{x}	√	×	{x, y}	√
4	∅	{sum, x}	×	×	{sum, x, y}	×
5	{y}	{sum, x}	×	×	{sum, x}	√
6	∅	{sum, x}	×	×	{sum, x, y}	×
7	∅	{sum, x}	×	√	{sum, x, y}	√
8	{sum}	{sum, x}	√	×	{sum, x, y}	√
9	{y}	{sum, x}	×	×	{sum, x, y}	√
10	∅	{sum}	×	×	{sum}	×
11	∅	{sum}	×	×	{sum}	×
		{sum}			{sum}	

m	DEF(m)	Cond ₂	Rel ₂	Slice ₂	Slice ^(*)
1	{x}	×	∅	✓	✓
2	{y}	×	{x}	✓	✓
3	{sum}	×	{x, y}	✓	✓
4	∅	✓	{sum, x, y}	✓	✓
5	{y}	×	{sum, x}	✓	✓
6	∅	×	{sum, x, y}	×	✓
7	∅	✓	{sum, x, y}	✓	✓
8	{sum}	×	{sum, x, y}	✓	✓
9	{y}	×	{sum, x, y}	✓	✓
10	∅	×	{sum}	×	✓
11	∅	×	{sum}	×	×

(*) Syntactic check after generating the slice:

if then (/else) ∈ Slice ⇒ (the corresponding) **end if** ∈ Slice

while ... do ∈ Slice ⇒ (the corresponding) **end while** ∈ Slice

...

The Ideal Slicing Algorithm?

Slice wrt. $(2, \{x\})$?

1: Input(x)

2: $x := x$

The Ideal Slicing Algorithm?

Slice wrt. $(2, \{x\})$?

- 1: Input(x)
- 2: $x := x$

Slice wrt. $(5, \{x\})$?

- 1: **if** true **then**
- 2: $x := 1$
- 3: **else**
- 4: $x := 2$
- 5: **end if**

The Ideal Slicing Algorithm?

Slice wrt. $(2, \{x\})$?

- 1: Input(x)
- 2: $x := x$

Slice wrt. $(5, \{x\})$?

- 1: **if** true **then**
- 2: $x := 1$
- 3: **else**
- 4: $x := 2$
- 5: **end if**

No algorithm for the **smallest slice** exists!

Reason: **Undecidability** of halting/termination.

Slicing: Applications

1. Test adequacy: for each output variable, all du-paths in its slice must be covered
2. Robustness testing: Add pseudo-variables that check dangerous situations, generate the slice and test
3. Regression testing: testing if a change influences a particular component (i.e., if the slice of the component interface contains the change)
4. Debugging:
code review
comparing a correct running program with a new faulty version

Outline

Debugging

Sequential Slicing

Structured Slicing

Automated Debugging

Simplifying the Test-Case

Isolating the Causes

Automated Debugging is about Perfection

Perfection

*Perfection is achieved not when you have nothing more to add, but when there is **nothing more left to take away**.*

Antoine de Saint-Exupéry

Automated Debugging is about Perfection

Perfection

*Perfection is achieved not when you have nothing more to add, but when there is **nothing more left to take away**.*

Antoine de Saint-Exupéry

Automated Debugging

Take out all that has nothing to do with the **failure**...

Debugging: An Example

- ▶ My slides for today (in \LaTeX) did not compile
- ▶ some part of it did work before (older slides)

Debugging: An Example

- ▶ My slides for today (in \LaTeX) did not compile
- ▶ some part of it did work before (older slides)
- ▶ divide the new parts into two:
 1. remove first half part
 2. if the problem is there, repeat until one (new) slide is left
 3. if not, put back the second half and remove the first, repeat
- ▶ apply the same technique to the content of the remaining slide

Debugging: An Example

- ▶ My slides for today (in \LaTeX) did not compile
- ▶ some part of it did work before (older slides)
- ▶ divide the new parts into two:
 1. remove first half part
 2. if the problem is there, repeat until one (new) slide is left
 3. if not, put back the second half and remove the first, repeat
- ▶ apply the same technique to the content of the remaining slide

This is called **delta debugging**:
our order of business for today.

Outline

Debugging

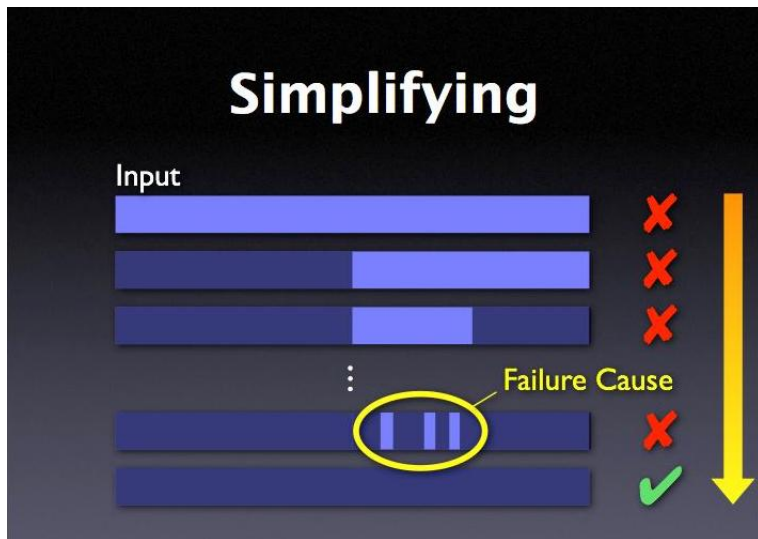
Sequential Slicing

Structured Slicing

Automated Debugging

Simplifying the Test-Case

Isolating the Causes



(Ack. figures are due to Andreas Zeller.)

Minimizing Delta Debugging: Basic Idea

Try to find the minimal environment causing the failure by:

- ▶ Divide the circumstances C in n parts C_i ,
- ▶ remove a part C_i such that $C \setminus C_i$ causes failure, repeat the algorithm with $C \setminus C_i$,
- ▶ if no such part exists, choose a bigger $n < |C|$ and repeat.

Minimizing Delta Debugging: Formalization

- ▶ Circumstances: C (input but could be: program, environment, etc.)
- ▶ Test: $test : 2^C \rightarrow \{\times, \checkmark, ?\}$
- ▶ Starting state: $C_x \subseteq C$, such that $test(C_x) = \times$
- ▶ Goal: find a **minimal subset** $C'_x \subseteq C_x$ such that $test(C'_x) = \times$

Minimizing Delta Debugging: Algorithm

$ddmin(C_x, 2)$, where

$ddmin(C'_x, n) =$

$$\begin{cases} C'_x, & \text{if } |C'_x| = 1, \\ ddmin(C'_x \setminus C_i, \max(n-1, 2)) & \text{else if } \exists_{i \leq n} \text{test}(C'_x \setminus C_i) = \times \\ ddmin(C'_x, \max(2n, |C'_x|)) & \text{else if } n < |C'_x| \\ C'_x & \text{otherwise} \end{cases}$$

where C_i 's are partitions of C'_x of (almost) equal size.

Application in Random Testing

Idea

- ▶ feed huge inputs to the system
(guaranteed crash on huge input)
- ▶ simplify input
- ▶ present the simplified result as a test-case

Application in Random Testing

Examples

- ▶ applied to command UNIX tools
- ▶ FLEX (lexical analyzer): crashed on a test-case of 2121 characters
- ▶ NROFF (document formatter): crashed on a single control character
- ▶ CRTPLOT (plotter output): crashed on single characters 't' or 'f'

Improvements

- ▶ caching: **save** the test outcomes, use the saved data
- ▶ stop early: define a **criterion to stop** the algorithm, e.g.,
 1. no progress
 2. reaching a certain granularity
 3. upper bound on time
- ▶ use structures, e.g., blocks instead of characters
- ▶ differences vs. circumstances (compare sane with insane)

Outline

Debugging

Sequential Slicing

Structured Slicing

Automated Debugging

Simplifying the Test-Case

Isolating the Causes

What is a Cause?

- ▶ Effect: the failure
- ▶ Cause: an event **preceding** effect,
without which effect would **not** have happened

Isolating the cause

- ▶ Cause: the **minimal difference** between the worlds with and **without the failure**

Isolating the cause

- ▶ Cause: the **minimal difference** between the worlds with and **without the failure**
- ▶ Challenge: the world **without failure**: the goal of debugging

Isolating the cause

- ▶ Cause: the **minimal difference** between the worlds with and **without the failure**
- ▶ Challenge: the world **without failure**: the goal of debugging
- ▶ Two solutions:
 1. **manipulate** the world by a **debugger**: turn infected to sane
 2. use **another test-case** in which no fault appears

Isolating: The Sorting Program Case

1. ./sample produces a **failure** on 5 4 3 666666
2. works **fine** on 5 4 3
3. find combinations of
 - 3.1 states of **1 with 2** such that the program **passes**
 - 3.2 states of **2 with 1** such that the program **fails**
4. the **difference** between the two leads to a **cause**

Isolating

Input



Delta Debugging: The Algorithm

Start from:

- ▶ $C_{\checkmark} = \emptyset$: passing circumstances and
 - ▶ C_{\times} : failing circumstances
1. compute the **difference** Δ between the failing and the passing circ., **divide** into n parts: Δ_j ,

Delta Debugging: The Algorithm

Start from:

- ▶ $C_{\checkmark} = \emptyset$: passing circumstances and
 - ▶ C_{\times} : failing circumstances
1. compute the **difference** Δ between the failing and the passing circ., **divide** into n parts: Δ_j ,
 2. **remove** Δ_j from the **failing** circ.; it is the new **passing** circ., if it **passes**

Delta Debugging: The Algorithm

Start from:

- ▶ $C_{\checkmark} = \emptyset$: passing circumstances and
 - ▶ C_{\times} : failing circumstances
1. compute the **difference** Δ between the failing and the passing circ., **divide** into n parts: Δ_j ,
 2. **remove** Δ_j from the **failing** circ.; it is the new **passing** circ., if it **passes**
 3. **add** Δ_j to the **passing** circ.; it is the new **failing** circ., if it **fails**
 4. **add** Δ_j to the **passing** circ.; it is the new **passing** circ., if it **passes**
 5. **remove** Δ_j from the **failing** circ.; it is the new **failing** circ., if it **fails**

Delta Debugging: The Algorithm

Start from:

- ▶ $C_{\checkmark} = \emptyset$: passing circumstances and
 - ▶ C_{\times} : failing circumstances
1. compute the **difference** Δ between the failing and the passing circ., **divide** into n parts: Δ_i ,
 2. **remove** Δ_i from the **failing** circ.; it is the new **passing** circ., if it **passes**
 3. **add** Δ_i to the **passing** circ.; it is the new **failing** circ., if it **fails**
 4. **add** Δ_i to the **passing** circ.; it is the new **passing** circ., if it **passes**
 5. **remove** Δ_i from the **failing** circ.; it is the new **failing** circ., if it **fails**
 6. **increase** n if **none** of the above holds

Delta Debugging: The Algorithm

Start from:

- ▶ $C_{\checkmark} = \emptyset$: passing circumstances and
 - ▶ C_{\times} : failing circumstances
1. compute the **difference** Δ between the failing and the passing circ., **divide** into n parts: Δ_i ,
 2. **remove** Δ_i from the **failing** circ.; it is the new **passing** circ., if it **passes**
 3. **add** Δ_i to the **passing** circ.; it is the new **failing** circ., if it **fails**
 4. **add** Δ_i to the **passing** circ.; it is the new **passing** circ., if it **passes**
 5. **remove** Δ_i from the **failing** circ.; it is the new **failing** circ., if it **fails**
 6. **increase** n if **none** of the above holds
 7. **repeat** until the difference is a **singleton**

Delta Debugging: Algorithm

$dd(C_{\checkmark}, C_{\times}, 2)$,

where $ddmin(C'_{\checkmark}, C'_{\times}, n)$ is defined recursively as:

$$\left\{ \begin{array}{ll} (C'_{\checkmark}, C'_{\times}) & \text{if } |\Delta| = 1, \\ dd(C'_{\times} \setminus \Delta_i, C'_{\times}, 2) & \text{else if } \exists_{i \leq n} \text{test}(C'_{\times} \setminus \Delta_i) = \checkmark \\ dd(C'_{\checkmark}, C'_{\checkmark} \cup \Delta_i, 2) & \text{else if } \exists_{i \leq n} \text{test}(C'_{\checkmark} \cup \Delta_i) = \times \\ dd(C'_{\checkmark} \cup \Delta_i, C'_{\times}, \max(n-1, 2)) & \text{else if } \exists_{i \leq n} \text{test}(C'_{\checkmark} \cup \Delta_i) = \checkmark \\ dd(C'_{\checkmark}, C'_{\times} \setminus \Delta_i, \max(n-1, 2)) & \text{else if } \exists_{i \leq n} \text{test}(C'_{\times} \setminus \Delta_i) = \times \\ dd(C'_{\checkmark}, C'_{\times}, \min(2n, |\Delta|)) & \text{else if } n < |\Delta| \\ (C'_{\checkmark}, C'_{\times}) & \text{otherwise} \end{array} \right.$$

where $\Delta = C'_{\times} \setminus C'_{\checkmark}$ and Δ_i 's are n partitions of Δ of (almost) equal size.

Delta Debugging: Applied to Test-Case Simplification

Start from:

- ▶ $C_{\checkmark} = \emptyset$: the empty test-case
- ▶ C_{\times} : the test-case leading to failure
- ▶ Much more efficient than minimizing delta debugging

Delta Debugging: Applied to Regression Testing

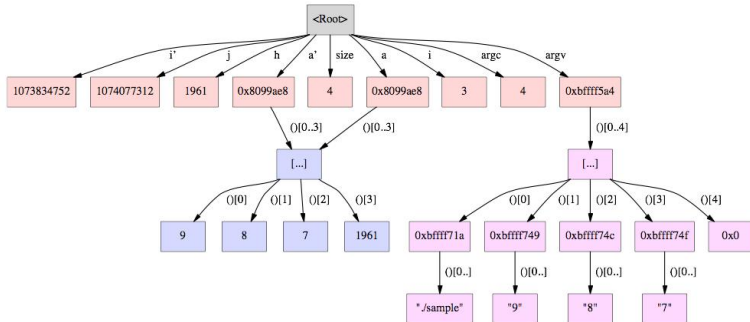
Start from:

- ▶ Goal: find out what went wrong in the new development (the old version worked well)
- ▶ $C_{\checkmark} = \emptyset$: basis is the old program, no changes needed
- ▶ C_{\times} : difference between the old and the new
i.e., changes needed to obtain the new program from the old one

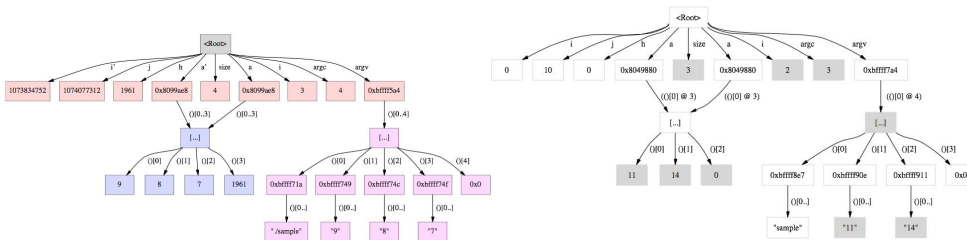
Isolating the Cause: Idea

- ▶ Capture the state of the program
- ▶ Compare the states of a passes and a failed run
- ▶ The smallest difference Δ is the variable causing the problem
- ▶ Find out what influences this variable

Program State: Memory Graphs



Comparing the Differences



Implementable as debugger commands,
e.g., set variable size = 2.

Isolating the Cause: Implementation

- ▶ Compute the **common subgraph** of the passing and failing memory graphs. Let the difference be C_x .
- ▶ Implement C_x as **debugger commands**.
- ▶ Apply delta-debugging to $C_v = \emptyset$ and C_x
 1. Apply differences to the memory graphs and test.
 2. At each step of dd if the changed state is not a valid state (program does not run), return ?, if it is a valid state, return the result of the test,
- ▶ The result Δ leads to a cause.

Isolating the Cause: Sorting Case

Run the algorithm before calling `shell_sort` with the state of `./sample 7 8 9` as passing and `./sample 11 14` as failing.

If 0 at the state: test fails \times , passes \checkmark otherwise.

1. $C_{\times} = \{ a[], i, size, argc, argv[] \}$, $C_{\checkmark} = \emptyset$.
2. new failing state: `a[], argv[1]` \times
3. new passing state: `argv[1]` \checkmark
4. new passing state: `a[0]` \checkmark
5. new passing state: `a[0]` and `a[1]` \checkmark
6. $\Delta = \{ a[2] \}$

Isolating the Cause: Illustrated Case

■ = δ is applied, □ = δ is *not* applied

#	$a'[0]$	$a[0]$	$a'[1]$	$a[1]$	$a'[2]$	$a[2]$	$argc$	$argv[1]$	$argv[2]$	$argv[3]$	i	$size$	Output	Test
1	□	□	□	□	□	□	□	□	□	□	□	□	7 8 9	✓
2	■	■	■	■	■	■	■	■	■	■	■	■	0 11	✗
3	■	■	■	■	■	■	□	□	□	□	□	□	0 11 14	✗
4	■	■	■	□	□	□	□	□	□	□	□	□	7 11 14	?
5	□	□	□	■	■	■	□	□	□	□	□	□	0 9 14	✗
6	□	□	□	■	□	□	□	□	□	□	□	□	7 9 14	?
7	□	□	□	□	■	■	□	□	□	□	□	□	0 8 9	✗
8	□	□	□	□	■	□	□	□	□	□	□	□	0 8 9	✗
Result					■									

Isolating the Chain of Causes

- ▶ Apply delta-debugging at the start, determine the minimal passing and running state
- ▶ Choose a common point (e.g., a function call) in the middle
- ▶ Apply delta-debugging on the states of the minimal passing and failing run
- ▶ Repeat the algorithm with the rest of the program and the new passing and failing states

Finding the Culprits

- ▶ The previous algorithm gives different Δ 's (causes at different points)
- ▶ Track the change of causes
- ▶ A smelling point: *a* ceases to be a cause and *b* becomes a cause

Automated Debugging

- ▶ A natural mechanization of simple debugging principles
- ▶ Provides (partial) solutions to
 1. testing,
 2. simplifying the test-cases,
 3. isolating the causes and
 4. isolating the cause-effect chain.

Notes on the Reading Material

- ▶ Covered: Chapters 5, 13 (apart from 13.6) and 14
- ▶ Chapters 1 and 12 provide background information
- ▶ Andreas Zeller's slides are also a very good source (see web page)
- ▶ Igor command-line tool can be downloaded from www.askigor.org
(unfortunately, the debugging web-service is closed by now)