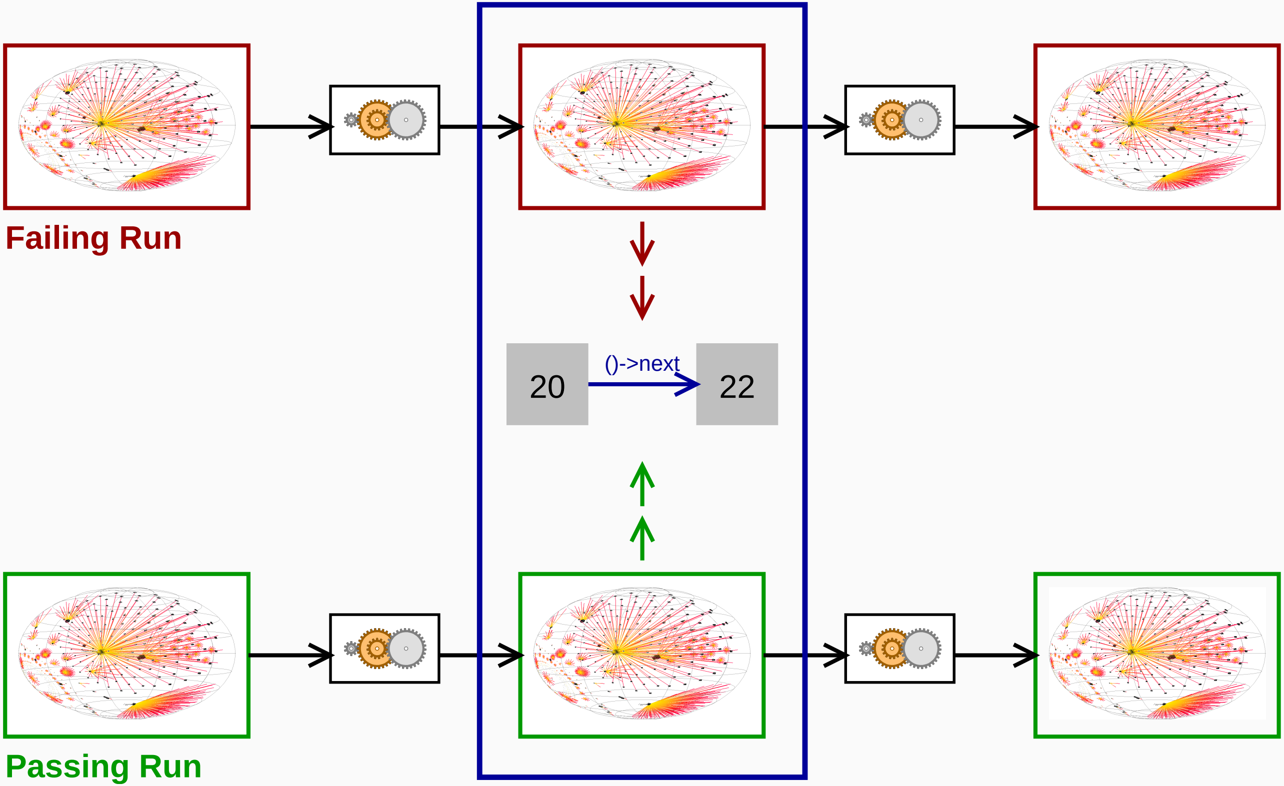
Isolating Cause-Effect Chains II

Andreas Zeller

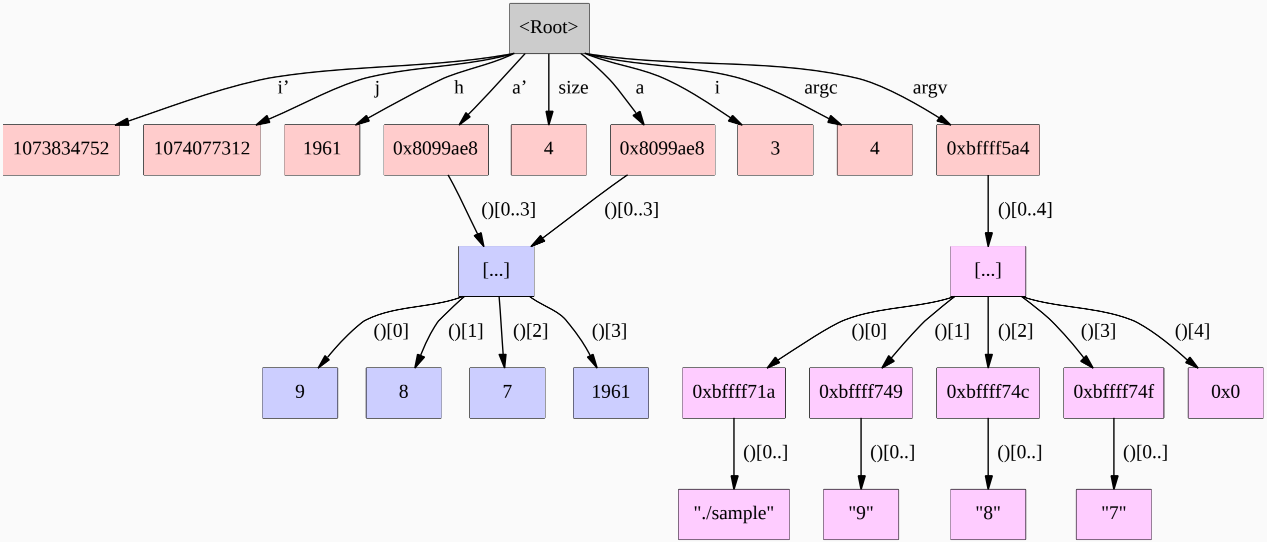
Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

The Process in a Nutshell



Memory Graphs

Let $G = (V, E, root)$ be a memory graph containing a set V of vertices, a set E of edges, and a dedicated vertex *root*.



Vertices

Each vertex $v \in V$ has the form $v = (val, tp, addr)$, standing for a value val of type tp at memory address $addr$.

As an example, the C declaration

```
int i = 42;
```

results in a vertex $v_{\mathtt{i}} = (42, \text{int}, \textit{0x1234})$, where *0x1234* is the (hypothetical) memory address of `i`.

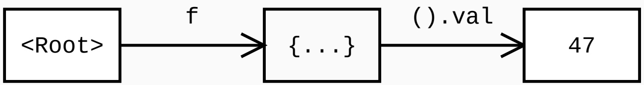
Edges

Each edge $e \in E$ has the form $e = (v_1, v_2, op)$, where $v_1, v_2 \in V$ are the related vertices. The operation op is used in constructing the expression of a vertex.

Example:

```
struct foo { int val; } f = {47};
```

results in two vertices
$v_f = (\{\ldots\}, \text{struct foo}, \mathit{0x5678})$ and
$v_{f.val} = (47, \text{int}, \mathit{0x9abc})$, as well as an edge
$e_{f.val} = (v_f, v_{f.val}, op_{f.val})$ from v_f to $v_{f.val}$



Root

A memory graph contains a dedicated vertex $root \in V$ that references all base variables of the program. Each vertex in the memory graph is accessible from root.

Example:

```
int i = 42;
struct foo { int val; } f = {47};
```

`i` and `f` are base variables; thus, the graph contains the edges $e_i = (root, v_i, op_i)$ and $e_f = (root, v_f, op_f)$.

Operations

Edge operations construct the name of descendants from their parent's name.

In an edge $e = (v_1, v_2, op)$, each operation op is a *function* that takes the expression of v_1 to construct the expression of v_2.

We denote functions by $\lambda x.B$—a function that has a formal parameter x and a body B.

In our examples, B is simply a string containing x; applying the function returns B where x is replaced by the function argument.

Operations (2)

Example:

```
int i = 42;
struct foo { int val; } f = {47};
```

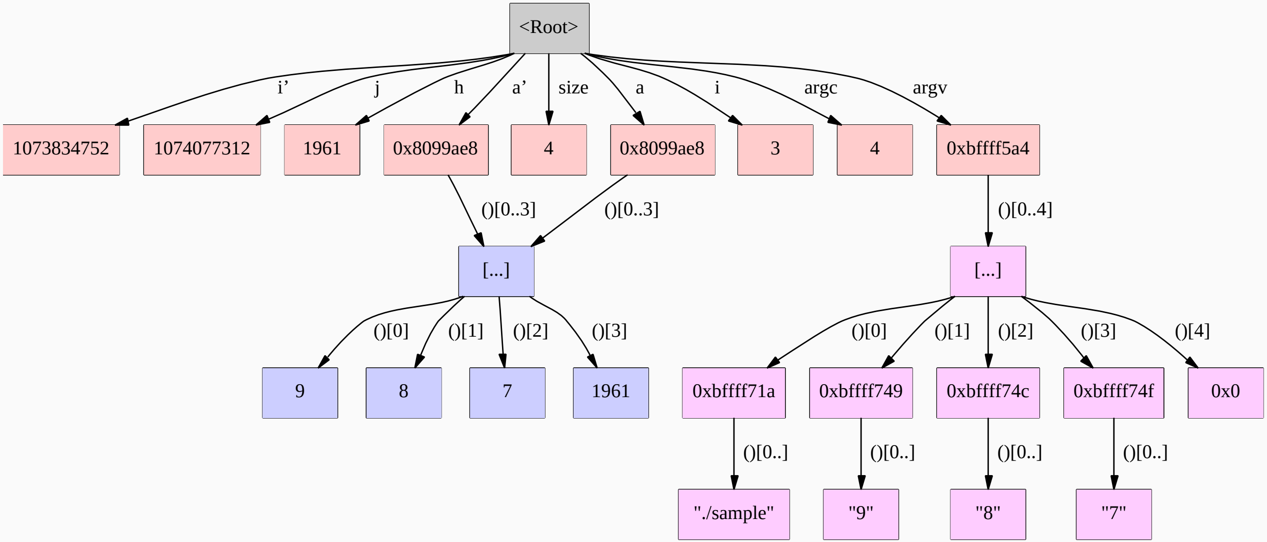
Operations on edges leading from *root* to base variables initially set the name; so $op_i = \lambda x.$"i" and $op_f = \lambda x.$"f" hold.

Deeper vertices are constructed based on the name of their parents.

Example: $op_{f.val} = \lambda x.$"x.val"
"to access the name of the descendant, one must append ".val" to the name of its parent".

Operations (3)

In visualizations, the operation body is shown as *edge label*. The formal parameter is replaced by "()" (formally: the label is *op*("()")).



Names

The function *name* constructs a name for a vertex v using the operations on the path from v to the root vertex.

$$name(v) = \begin{cases} op(name(v')) & \text{if } \exists(v', v, op) \in E \\ \texttt{""} & \text{otherwise (root vertex)} \end{cases}$$

Example—a name for $v_{f.val}$:

$$\begin{aligned} name(v_{f.val}) &= op_{f.val}(name(v_f)) \\ &= op_{f.val}(op_f(\texttt{""})) \\ &= op_{f.val}(\texttt{"f"}) \\ &= \texttt{"f.val"} \end{aligned}$$

A vertex can have multiple names (“aliasing”).

Unfolding Memory Graphs

1. Let $unfold(parent, op, G)$ be a procedure (sketched below) that takes the name of a parent expression *parent* and an operation *op* and unfolds the element $op(parent)$, adding new edges and vertices to the memory graph G.
2. Initialize $V = \{root\}$ and $E = \emptyset$.
3. For each base variable *name* in the program, invoke $unfold(root, \lambda x."name")$.

Unfolding Aliases

Let $(V, E, root) = G$ be the members of G, let $expr = op(parent)$ be the expression to unfold, let tp be the type of $expr$, and let $addr$ be its address.

If V already has a vertex v' at the same address and with the same type (formally, $\exists v' = (val', tp', addr') \in V \cdot tp = tp' \wedge addr = addr'$), do not unfold $expr$ again; however, insert an edge $(parent, v', op)$ to the existing vertex.

Example:

```
struct foo f; int *p1; int *p2; p1 = p2 = &f;
```

If `f` has already been unfolded, we do not need to unfold its aliases `*p1` and `*p2`. However, we insert edges from `p1` and `p2` to `f`.

Unfolding Records

Let $(V, E, root) = G$ be the members of G, let $expr = op(parent)$ be the expression to unfold, let tp be the type of $expr$, and let $addr$ be its address.

If $expr$ is a record containing n members $m_1, m_2, \ldots, m_n$, add a vertex $v = (\{\ldots\}, tp, addr)$ to V, and an edge $(parent, v, op)$ to E.

For each $m_i \in \{m_1, m_2, \ldots, m_n\}$, invoke $unfold(expr, \lambda x.\text{"}x\,.\,m_i\text{"}, G)$, unfolding the record members.

Unfolding Arrays

Let $(V, E, root) = G$ be the members of G, let $expr = op(parent)$ be the expression to unfold, let tp be the type of $expr$, and let $addr$ be its address.

If $expr$ is an array containing n members $m[0], m[1], \ldots, m[n-1]$, add a vertex $v = ([\ldots], tp, addr)$ to V, and an edge $(parent, v, op)$ to E.

For each $i \in \{0, 1, \ldots, n\}$, invoke $unfold(expr, \lambda x.\texttt{"}x\texttt{[}i\texttt{]"}, G)$, unfolding the array elements.

Unfolding Pointers

Let $(V, E, root) = G$ be the members of G, let $expr = op(parent)$ be the expression to unfold, let tp be the type of $expr$, and let $addr$ be its address.

If $expr$ is a pointer with address value val, add a vertex $v = (val, tp, addr)$ to V, and an edge $(parent, v, op)$ to E.

Invoke $unfold(expr, \lambda x.\text{"*(}x\text{)"}, G)$, unfolding the element $expr$ points to (assuming that $*p$ is the dereferenced pointer p),

Unfolding Values

Let $(V, E, root) = G$ be the members of G, let $expr = op(parent)$ be the expression to unfold, let tp be the type of $expr$, and let $addr$ be its address.

If $expr$ contains an atomic value val, add a vertex $v = (val, tp, addr)$ to V, and an edge $(parent, v, op)$ to E.

What does P point to?

In C, uninitialized pointers can contain arbitrary addresses. A pointer referencing invalid or uninitialized memory can quickly introduce lots of garbage into the memory graph.

To distinguish valid from invalid pointers, we use a *memory map* consisting of *memory areas* like

- stack frames
- heap areas requested via the *malloc* function
- known variables in static memory

A pointer is valid only if it points within a known area.

Sample Memory Map

Static area

```
[0x080480f4 - 0x0804982c] (5944 bytes) static
[0x08048718 - 0x08048726] (14 bytes) static .rodata
[0x08049728 - 0x08049734] (12 bytes) static .data
[0x08049774 - 0x08049814] (160 bytes) static .dynamic
[0x08049814 - 0x0804982c] (24 bytes) static .bss
```

Stack area

```
[0xbffff464 - 0xc0000000] (2972 bytes) stack
[0xbffff464 - 0xbffff48c] (40 bytes) stack frame #0
[0xbffff48c - 0xbffff4bc] (48 bytes) stack frame #1
[0xbffff4bc - 0xbffff4f8] (60 bytes) stack frame #2
[0xbffff4f8 - 0xc0000000] (2824 bytes) args+env
```

Heap area

```
[0x0804982c - 0x0804984c] (32 bytes) malloc
[0x08049838 - 0x0804984c] (20 bytes) malloc
```

Dynamic Arrays

In C, one can allocate arrays of arbitrary size on the heap via the *malloc* function.

While the base address of the array is typically stored in a pointer, C offers no means to find out how many elements were actually allocated.

Basic idea: use the memory map; the referred elements cannot extend beyond area boundaries.

Example—we know that an array lies within a memory area of 1000 bytes. The array cannot be longer than 1000 bytes.

Unions

In C, unions (also known as variant records) allow multiple types to be stored at the same memory address. Keeping track of the actual type is left to the discretion of the programmer.

Basic idea: disambiguate unions by *weirdness*—choose the alternative with the least amount of

- invalid pointers
- invalid strings
- invalid enumeration elements
- unused memory

Unions (2)

Example:

```
union {
    void *ptr;      // 0xdeadbeef (invalid)
    char str[4];    // "ï¾Þ"
    int num;        // 3735928559
} u;
```

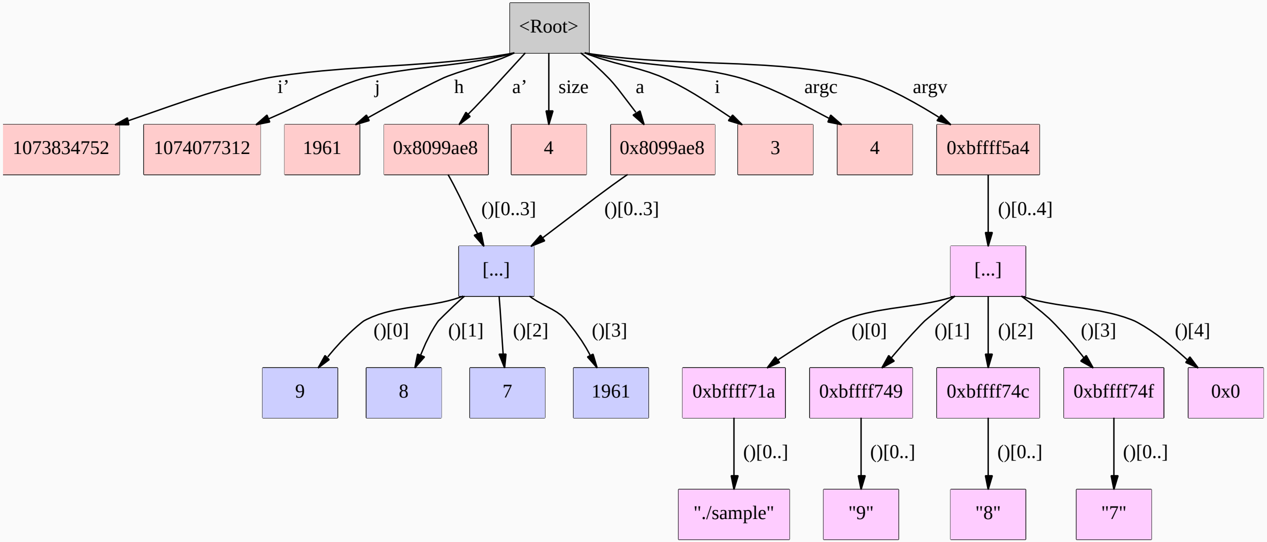
`u.ptr` is weird (invalid)

`u.str` contains weird characters

`u.num` is the *least weird alternative*

All Things Considered

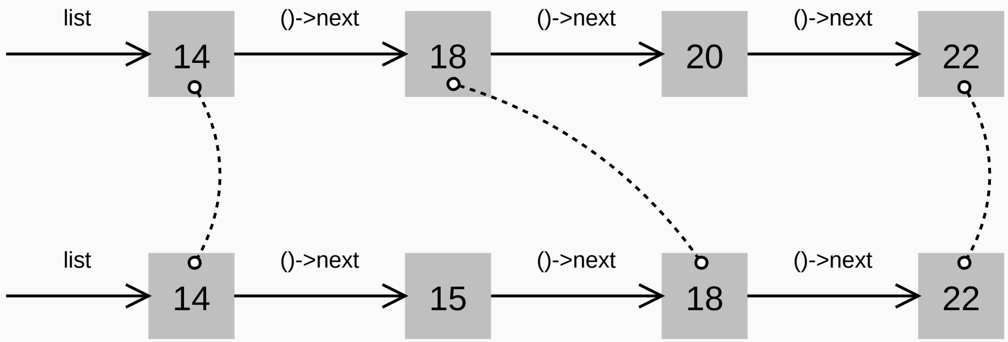
With all these heuristics, we eventually obtain quite accurate memory maps:



However, *tracking* union and pointer assignments (rather than guessing the most likely alternative) would be more accurate.

Comparing Memory Graphs

As a human, you can quickly grasp *differences* between small graphs:



To detect such differences automatically, though, requires some graph operations.

Comparing Memory Graphs (2)

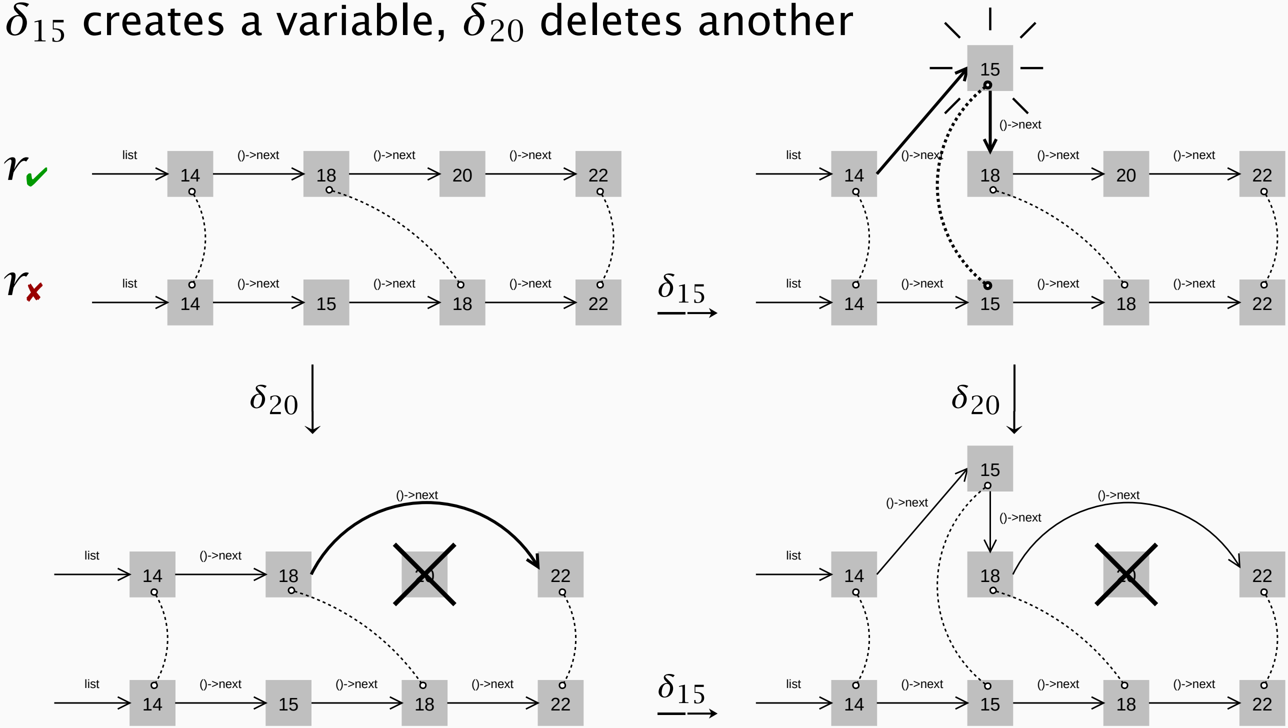
Basic idea: compute the *maximum common subgraph*—the greatest possible *matching* between the two graphs. Formally:

1. Create the set of all pairs of vertices (v_1, v_2) with the same value and the same type, one from each graph. Formally, $v_1 \in V_1$, $v_2 \in V_2$ and $val_1 = val_2 \wedge tp_1 = tp_2$ holds where $(val_1, tp_1, addr_1) = v_1$ and $(val_2, tp_2, addr_2) = v_2$.
2. Form the *correspondence graph* C whose nodes are the pairs from (1). Any two vertex pairs $v = (v_1, v_2)$ and $v' = (v'_1, v'_2)$ in C are connected if
   - the operations of the edges (v_1, v'_1, op_1) in G_1 and (v_2, v'_2, op_2) in G_2 are the same, i.e. $op_1 = op_2$, or
   - neither (v_1, v'_1, op_1) nor (v_2, v'_2, op_2) exist.
3. The maximal common subgraph then corresponds to the *maximum clique* in C—that is, a complete subgraph of C that is not contained in any other complete subgraph.

Exponential in the worst case (= no labels, all contents are equal); use *heuristics* as alternative.

Comparing Memory Graphs (3)

The common subgraph induces *structural graph differences:*
δ_{15} creates a variable, δ_{20} deletes another



External State

Memory graphs can easily model external state, simply by adding appropriate *operations.*

File system. A *file descriptor* refers to a *file* with *content* and *attributes* (e.g. *name*, *attributes*, . . .)

User interface. A *window handle* refers to a *window* with *content* and *attributes* (e.g. *title*. *size*, *position*, . . .)

Internet. A *URL* refers to a *document* with *content* and even more URLs. . .

Note that all "external" states are eventually read into "internal" states; hence, getting and comparing such states may not be required.

Limits of Delta Debugging

Delta Debugging has a number of potential *drawbacks:*

False Positives. What makes a failure a failure?

Local Minimum. Return first hit instead of the smallest.

State Artefacts. Results may not apply to original runs.

One Run. What do we do if we have only one run?

Isolate Infection. Result applies only on state, not on code.

False Positives


```
bool x = ...;
bool y = x;
// <= Access state here
if (x != y)
    fail();
if (x && y)
    fail();
```

Run	x	y
$r_{✓}$	false	false
$r_{✗}$	true	true
Tested r'	false	true

How do we distinguish the two `fail()` *calls?*

Approach: *compare more state*, e.g. backtraces

Local Minimum


```
bool x = read();
bool y = x;
// <= Access state here
if (x && y)
    fail();
```

Run	x	y
$r_{✔}$	false	false
$r_{✘}$	true	true
Tested r'	false	true

Should Delta Debugging return x, y, *or both?*

Current approach is *greedy*: *return first hit*

State Artefacts


```
bool a, b, c, d;
// <= Access state here
if ((a && !b && c) || (c && d))
    fail();
```

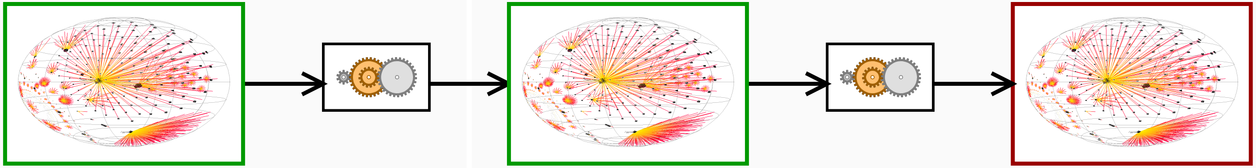
Run	a	b	c	d
$r_{✔}$	false	false	false	false
$r_{✘}$	true	true	true	true
Tested $r'_{✔}$	false	false	true	false
Tested $r'_{✘}$	true	false	true	false

The difference in a between $r'_{✔}$ and $r'_{✘}$ is failure-inducing. But altering a alone in $r_{✔}$ or $r_{✘}$ does not change the outcome!

Possible approach—*add more deltas until applicable to original run*

Working on a Single Run

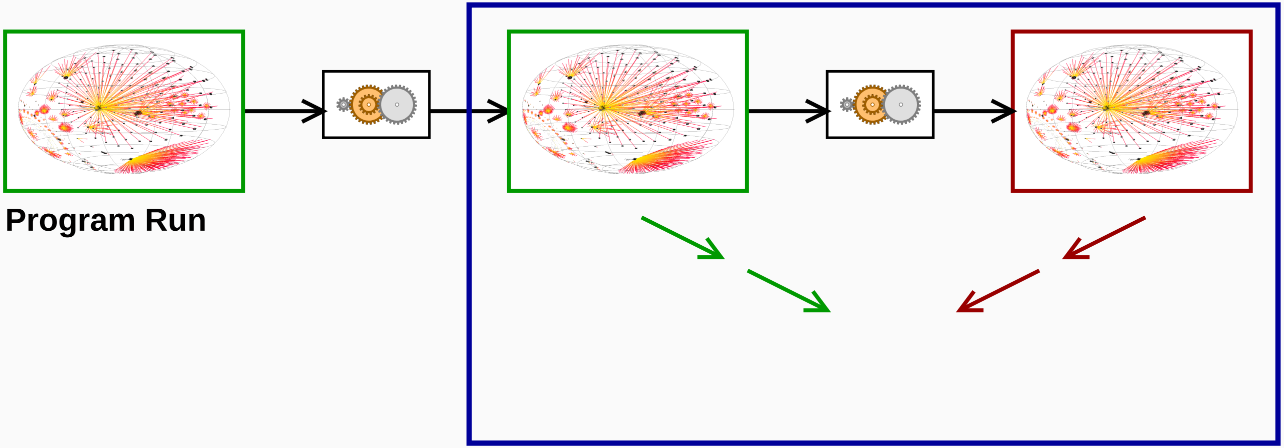
What do we do if we have only one failing run?



Program Run

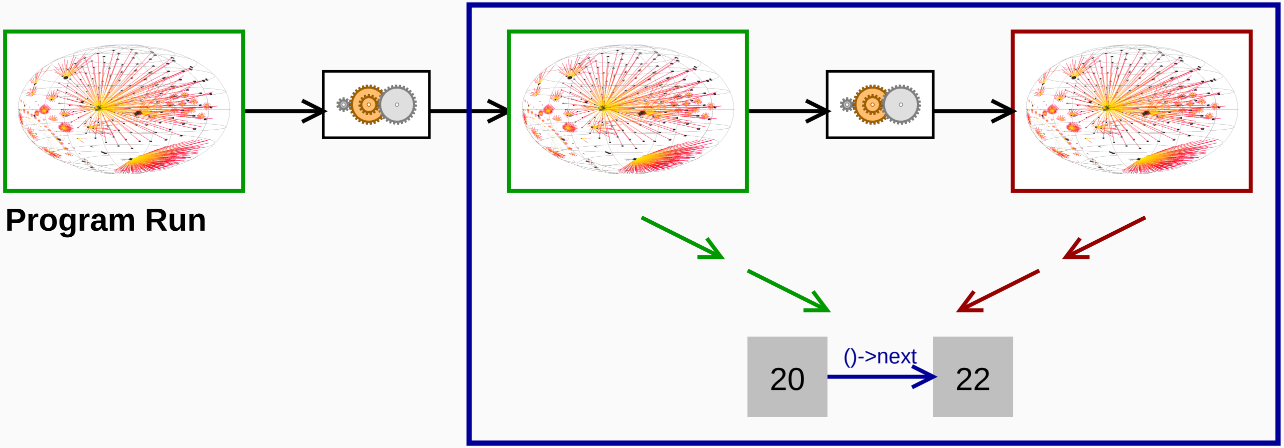
Working on a Single Run

What do we do if we have only one failing run?



Working on a Single Run

What do we do if we have only one failing run?



Requires that

- the test be applicable on *intermediate states*
- the states be *comparable* (i.e. same local variables → same backtrace → same program counter)

Single Run Example

Comparable states are typically obtained within *loops:*

```
while (1) {
    process_input();
    if (input_is_bad())    // <= Access state here
        fail();
}
```

Rather than fetching two complete runs $r_{\checkmark}$ and $r_{\times}$, work on two *loop iterations*—one where `fail()` is called and one where `fail()` is not called.

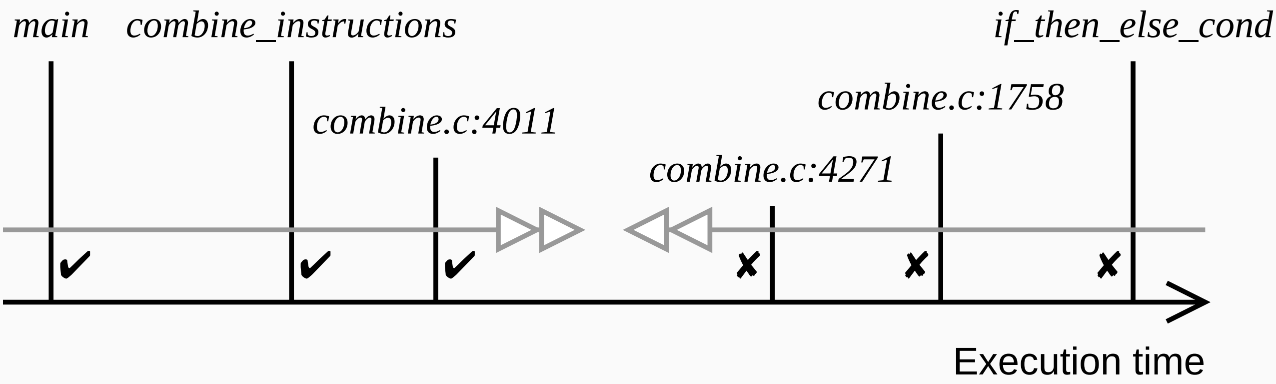
Not realized yet!

Narrowing Down Infection Sites

Delta Debugging only narrows down the current state.

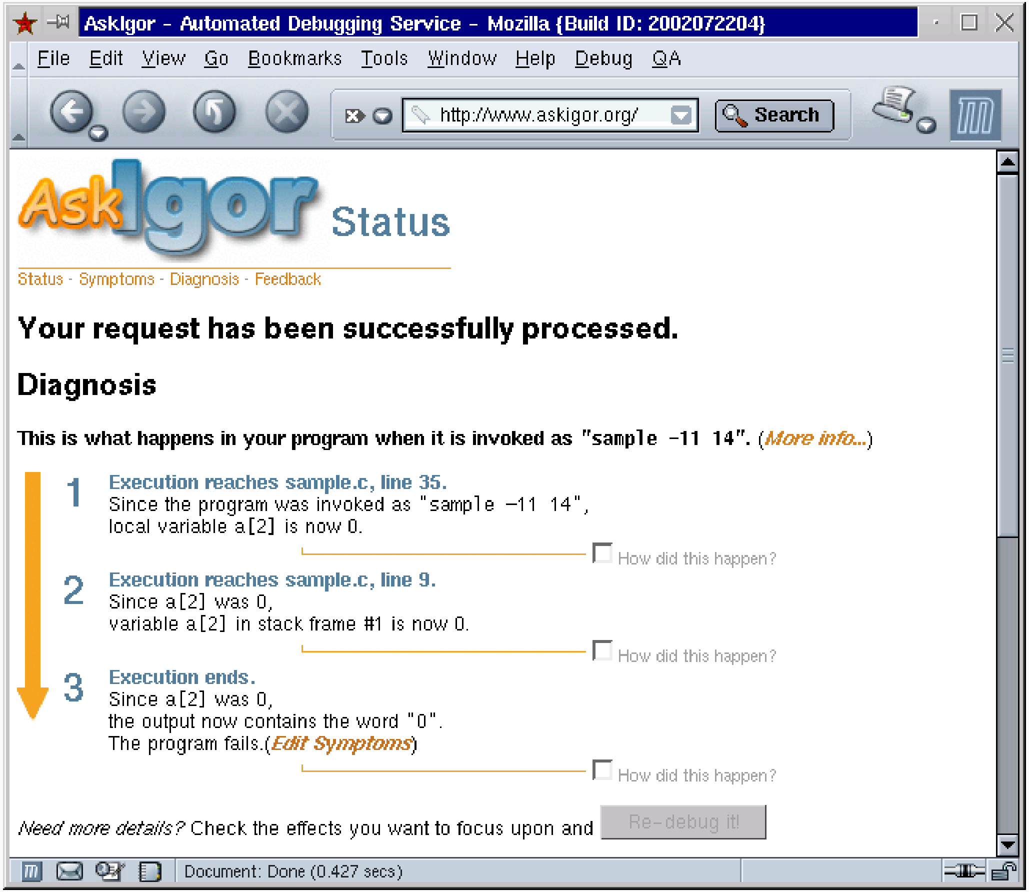
To narrow down the infection site, the programmer must still *assess the state* into “sane” (✔) and “infected” (✘).

Example—Narrowing down GCC infection:

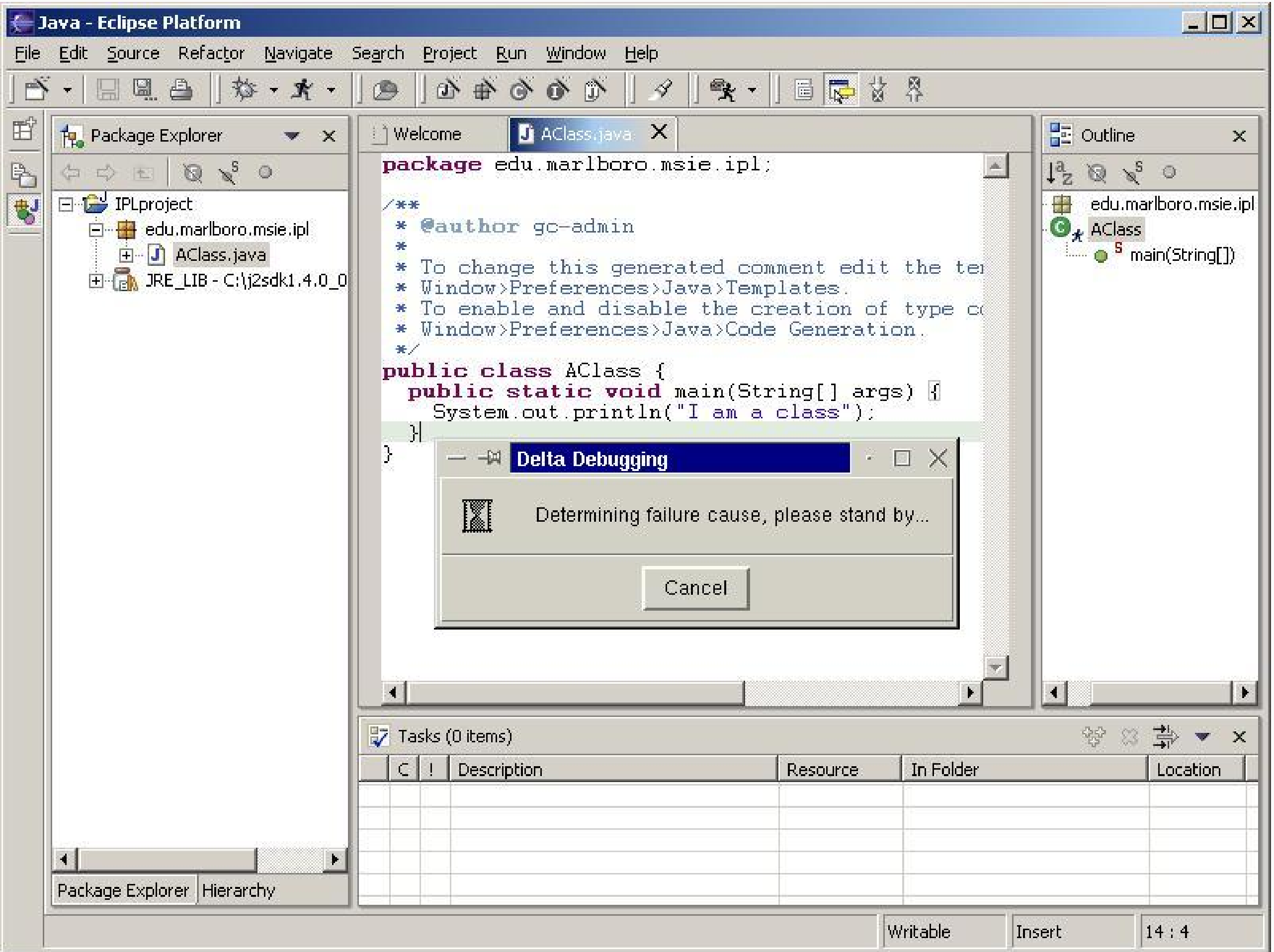


This temporal focusing can be done interactively!

How did this happen?



Master's Thesis, Anyone?



Concepts

➠ Memory graphs allow representing and comparing complex program states.

➠ Delta debugging has some open research questions:

- False Positives
- Local Minimum
- State Artefacts
- One Run

➠ Infection sites still must be narrowed down interactively

➠ Delta debugging functionality will soon be found in the top programming environments

References

- *Memory Graphs Web Site*,
  http://www.st.cs.uni-sb.de/memgraphs/
- T. Zimmermann, A. Zeller: *Visualizing Memory Graphs*, Proc. “Software Visualization” LNCS 2269, pp. 191–204,
  http://www.st.cs.uni-sb.de/papers/sv2001/
- *Delta Debugging Web Site*,
  http://www.st.cs.uni-sb.de/dd/
- *Eclipse Programming Environment*,
  http://www.eclipse.org/