Formal Definitions

A.1 DELTA DEBUGGING

A.1.1 Configurations

Definition A.1: *Configurations and runs.* We assume that the execution of a specific program is determined by a number of *circumstances.* Denote the set of possible *configurations* of circumstances by $\mathcal{R}$.

Definition A.2: $rtest$. The function $rtest: \mathcal{R} \to \{\text{✗}, \text{✓}, \text{?}\}$ determines for a program run $r \in \mathcal{R}$ whether some specific failure occurs (✗) or not (✓), or whether the test is unresolved (?).

Definition A.3: *Change.* A *change* δ is a mapping $\delta: \mathcal{R} \to \mathcal{R}$. The set of changes $\mathcal{C}$ is the set of all mappings from $\mathcal{R} \to \mathcal{R}$ (i.e., $\mathcal{C} = \mathcal{R}^{\mathcal{R}}$). The *relevant change* between two runs $r_1, r_2 \in \mathcal{R}$ is a change $\delta \in \mathcal{C}$ such that $\delta(r_1) = r_2$.

Definition A.4: *Composition of changes.* The *change composition* $\circ: \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as $(\delta_i \circ \delta_j)(r) = \delta_i\big(\delta_j(r)\big)$.

A.1.2 Passing and Failing Run

Axiom A.5: *Passing and failing run.* We assume two runs $r_{\text{✓}}, r_{\text{✗}} \in \mathcal{R}$ with $rtest(r_{\text{✓}}) = \text{✓}$ and $rtest(r_{\text{✗}}) = \text{✗}$.

In the following, we identify $r_{\text{✓}}$ and $r_{\text{✗}}$ by the changes applied to $r_{\text{✓}}$.

Definition A.6: $c_{\text{✓}}$. We define $c_{\text{✓}} \subseteq \mathcal{C}$ as the empty set $c_{\text{✓}} = \varnothing$, which identifies $r_{\text{✓}}$ (no changes applied).

Definition A.7: *Failing configuration.* The set of all changes $c_{\text{✗}} \subseteq \mathcal{C}$ is defined as $c_{\text{✗}} = \{\delta_1, \delta_2, \ldots, \delta_n\}$, identifying $r_{\text{✗}} = (\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(r_{\text{✓}})$.

A.1.3 Tests

Definition A.8: $test$. The function $test: 2^{c_{\text{✗}}} \to \{\text{✗}, \text{✓}, \text{?}\}$ is defined as follows: Let $c \subseteq c_{\text{✗}}$ be a configuration with $c = \{\delta_1, \delta_2, \ldots, \delta_n\}$. Then, $test(c) = rtest\big((\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(r_{\text{✓}})\big)$ holds.

Corollary A.9: *Passing and failing test case.* The following holds:

$$test(c_{\text{✓}}) = test(\varnothing) = rtest(r_{\text{✓}}) = \text{✓} \quad \text{and}$$
$$test(c_{\text{✗}}) = test\big(\{\delta_1, \delta_2, \ldots, \delta_n\}\big) = rtest(r_{\text{✗}}) = \text{✗}$$

A.1.4 Minimality

Definition A.10: *n-minimal configuration.* A configuration $c \subseteq c_{\text{✗}}$ is *n-minimal* if $\forall c' \subset c \cdot |c| - |c'| \leqslant n \Rightarrow (test(c') \neq \text{✗})$ holds.

Definition A.11: *Relevant configuration.* A configuration is called *relevant* if it is *1-minimal* in the sense of Definition A.10. Consequently, c is relevant if $\forall \delta_i \in c \cdot test(c \setminus \{\delta_i\}) \neq \text{✗}$ holds.

A.1.5 Simplifying

Proposition A.12: *ddmin minimizes.* For any $c \subseteq c_{\text{✗}}$, $ddmin(c)$ returns a relevant configuration in the sense of Definition A.11.

Proof. According to the *ddmin* definition (List 5.2), $ddmin(c'_{\text{✗}})$ returns $c'_{\text{✗}}$ only if $n \geqslant |c'_{\text{✗}}|$ and $test(\nabla_i) \neq \text{✗}$ for all $\Delta_1, \ldots, \Delta_n$ where $\nabla_i = c'_{\text{✗}} \setminus \Delta_i$. If $n \geqslant |c'_{\text{✗}}|$, then $|\Delta_i| = 1$ and $|\nabla_i| = |c| - 1$. Because all subsets of $c' \subset c'_{\text{✗}}$ with $|c'_{\text{✗}}| - |c'| = 1$ are in $\{\nabla_1, \ldots, \nabla_n\}$ and $test(\nabla_i) \neq \text{✗}$ for all ∇_i, the condition of Definition A.10 applies and c is 1-minimal. ■

Proposition A.13: *ddmin complexity, worst case.* The number of tests carried out by $ddmin(c_{\text{✗}})$ is $(|c_{\text{✗}}|^2 + 3|c_{\text{✗}}|)/2$ in the worst case.

Proof. The worst case can be divided into two phases. First, every test has an unresolved result until we have a maximum granularity of $n = |c_{\text{✗}}|$. Then, testing only the last complement results in a failure until $n = 2$ holds.

- In the first phase, every test has an unresolved result. This results in a reinvocation of $ddmin'$ with a doubled number of subsets, until $|c_i| = 1$ holds. The number of tests t to be carried out is $t = 2 + 4 + 8 + \cdots + |c_{\text{✗}}| = |c_{\text{✗}}| + \frac{|c_{\text{✗}}|}{2} + \frac{|c_{\text{✗}}|}{4} + \cdots = 2|c_{\text{✗}}|$.
- In the second phase, the worst case is that testing the *last* set $c'_{\text{✗}} \setminus \{c_n\}$ fails. Consequently, $ddmin'$ is reinvoked with $ddmin'(c'_{\text{✗}} \setminus \{c_n\})$. This results in $|c_{\text{✗}}| - 1$ calls of *ddmin*, with one test per call. The total number of tests t' is thus $t' = (|c_{\text{✗}}| - 1) + (|c_{\text{✗}}| - 2) + \cdots + 1 = 1 + 2 + 3 + \cdots + (|c_{\text{✗}}| - 1) = \frac{|c_{\text{✗}}|(|c_{\text{✗}}|-1)}{2} = \frac{|c_{\text{✗}}|^2 - |c_{\text{✗}}|}{2}$.

The overall number of tests is thus $t + t' = 2|c_{\text{✗}}| + (|c_{\text{✗}}|^2 - |c_{\text{✗}}|)/2 = (|c_{\text{✗}}|^2 + 3|c_{\text{✗}}|)/2$. ■

Proposition A.14: *ddmin complexity, best case.* If there is only one failure-inducing change $\delta_i \in c_{\text{✗}}$ and all configurations that include δ_i cause a failure as well, the number of tests t is limited by $t \leqslant \log_2(|c_{\text{✗}}|)$.

Proof. Under the given conditions, the test of either initial subset c_1 or c_2 will fail. $n = 2$ always holds. Thus, the overall complexity is that of a binary search. ■

A.1.6 Differences

Definition A.15: *n-minimal difference.* Let $c'_{\text{✓}}$ and $c'_{\text{✗}}$ be two configurations with $\varnothing = c_{\text{✓}} \subseteq c'_{\text{✓}} \subset c'_{\text{✗}} \subseteq c_{\text{✗}}$. Their difference $\Delta = c'_{\text{✗}} \setminus c'_{\text{✓}}$ is *n-minimal* if

$$\forall \Delta_i \subset \Delta \cdot |\Delta_i| \leqslant n \Rightarrow \left(test(c'_{\text{✓}} \cup \Delta_i) \neq \text{✓} \wedge test(c'_{\text{✗}} \setminus \Delta_i) \neq \text{✗}\right)$$

holds.

Definition A.16: *Relevant difference.* A difference is called *relevant* if it is *1-minimal* in the sense of Definition A.15. Consequently, a difference Δ is *1-minimal* if

$$\forall \delta_i \in \Delta \cdot test(c'_{\checkmark} \cup \{\delta_i\}) \neq \checkmark \wedge test(c'_{\times} \setminus \{\delta_i\}) \neq \times$$

holds.

A.1.7 Isolating

Proposition A.17: *dd minimizes.* Given $(c'_{\checkmark}, c'_{\times}) = dd(c_{\checkmark}, c_{\times})$, the difference $\Delta = c'_{\times} \setminus c'_{\checkmark}$ is 1-minimal in the sense of Definition A.15.

Proof. (compare proof of Proposition A.12): According to the *dd* definition (Figure 13.2), $dd'(c'_{\checkmark}, c'_{\times}, n)$ returns $(c'_{\checkmark}, c'_{\times})$ only if $n \geqslant |\Delta|$ where $\Delta = c'_{\times} \setminus c'_{\checkmark} = \Delta_1 \cup \cdots \cup \Delta_n$. That is, $|\Delta_i| = 1$ and $\Delta_i = \{\delta_i\}$ hold for all i.

Furthermore, for dd' to return $(c'_{\checkmark}, c'_{\times})$, the conditions $test(c'_{\checkmark} \cup \Delta_i) \neq \times$, $test(c'_{\times} \setminus \Delta_i) \neq \checkmark$, $test(c'_{\checkmark} \cup \Delta_i) \neq \checkmark$, and $test(c'_{\times} \setminus \Delta_i) \neq \times$ must hold.

These are the conditions of Definition A.15. Consequently, Δ is 1-minimal. ■

Proposition A.18: *dd complexity, worst case.* The number of tests carried out by $dd(c_{\checkmark}, c_{\times})$ is $|\Delta|^2 + 7|\Delta|$ in the worst case, where $\Delta = c_{\times} \setminus c_{\checkmark}$.

Proof. The worst case is the same as in Proposition A.13, but with a double number of tests. ■

Proposition A.19: *dd complexity, best case.* If all tests return either ✓ or ✗, the number of tests t in *dd* is limited by $t \leqslant \log_2(|c_{\times} \setminus c_{\checkmark}|)$.

Proof. We decompose $\Delta = \Delta_1 \cup \Delta_2 = c'_{\times} \setminus c'_{\checkmark}$. Under the given conditions, the test of $c'_{\checkmark} \cup \Delta_1 = c'_{\times} \setminus \Delta_2$ will either pass or fail. $n = 2$ always holds. This is equivalent to a classical binary search algorithm over a sorted array: with each recursion, the difference is reduced by $1/2$; the overall complexity is the same. ■

Corollary A.20: *Size of failure-inducing difference, best case.* Let $(c'_{\checkmark}, c'_{\times}) = dd(c_{\checkmark}, c_{\times})$. If all tests return either ✓ or ✗, then $|\Delta| = |c'_{\times} \setminus c'_{\checkmark}| = 1$ holds.

Proof. Follows directly from the equivalence to binary search, as shown in Proposition A.19. ■

A.2 MEMORY GRAPHS

A.2.1 Formal Structure

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* (Figure A.1):

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 $v_{\texttt{i}} = (42, \text{int}, 0x1234)$, where *0x1234* is the (hypothetical) memory address of `i`.

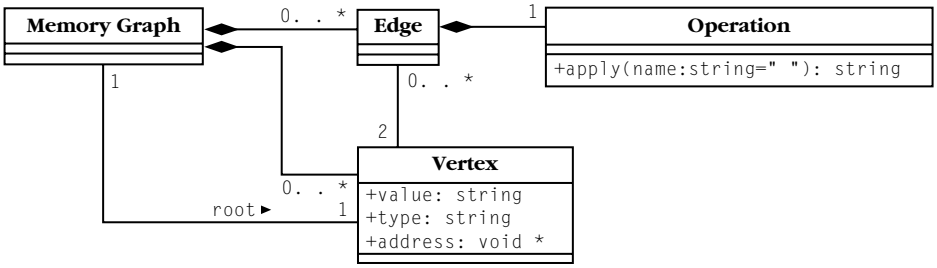

FIGURE A.1

UML object model of memory graphs.

FIGURE A.2

Edge construction.

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 (see Figure A.2). As an example, the C declaration of the record ("struct") `f`,

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

results in two vertices $v_f = (\{\ldots\}, \text{struct foo}, \textit{0x5678})$ and $v_{f.val} = (47, \text{int}, \textit{0x5678})$, 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 the root. In the previous examples, `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 on edges leading from *root* to base variables initially set the name. Thus, $op_i = \lambda x.$"`i`" and $op_f = \lambda x.$"`f`" hold.

Deeper vertices are constructed based on the name of their parents. For instance, $op_{f.val} = \lambda x.$"x `.val`" holds, meaning that to access the name of the descendant one must append "`.val`" to the name of its parent.

In our graph visualizations, the operation body is shown as edge label, with the formal parameter replaced by `"()"`. That is, we use *op*(`"()"`) as label. This is reflected in the previous figure.

Names. The following function *name* constructs a name for a vertex v using the operations on the path from v to the root vertex. As there can be several parents (and thus several names), we nondeterministically choose a parent v' of v along with the associated operation *op*.

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

As an example, see how a name for $v_{f.val}$ is found: $name(v_{f.val}) = op_{f.val}(name(v_f)) = op_{f.val}(op_f(\texttt{""})) = op_{f.val}(\texttt{"f"}) = \texttt{"f.val"}$.

A.2.2 Unfolding Data Structures

To obtain a memory graph $G = (V, E, root)$, as formalized in Section A.2.1, we use the following scheme.

1. Let *unfold*(*parent*, *op*, *G*) be a procedure (sketched in the following) 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 $E = \varnothing$.
3. For each base variable *name* in the program, invoke *unfold*(*root*, $\lambda x.$"*name*").

The *unfold* procedure works as follows. 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. The unfolding then depends on the structure of *expr*.

Aliases. 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*) in the existing vertex. As an example, consider the C statements:

```
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`.

Records. Otherwise, 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*, $\lambda x.$"$x.m_i$", *G*), unfolding the record members.

As an example, consider the "Edges" example shown in Figure A.2. Here, the record `f` is created as a vertex and its member `f.val` has been unfolded.

Arrays. Otherwise, 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 $(parent, v, op)$

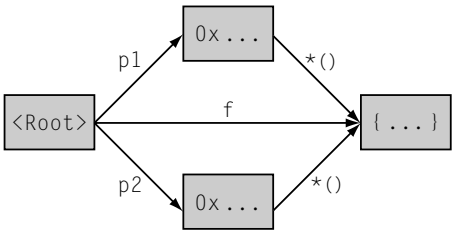

FIGURE A.3

Alias graph.

to E. For each $i \in \{0, 1, \ldots, n\}$, invoke $unfold(expr, \lambda x."x[i]", G)$, unfolding the array elements. Arrays are handled very much like records, and thus no example is given.

Pointers. Otherwise, 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* $(expr, \lambda x."*(x)", G)$, unfolding the element *expr* points to (assuming that $*p$ is the dereferenced pointer p). In the previous "Aliases" example, we would end up with the graph shown in Figure A.3.

Atomic values. Otherwise, *expr* contains an atomic value *val*. Add a vertex $v = (val, tp, addr)$ to V, and an edge $(parent, v, op)$ to E. As an example, see `f` in the previous figure.

A.2.3 Matching Vertices and Edges

Let $G_✓ = (V_✓, E_✓, root_✓)$ and $G_✗ = (V_✗, E_✗, root_✗)$ be two memory graphs.

Matching vertices. Two vertices $v_✓ \in V_✓$ and $v_✗ \in V_✗$ *match* (written $v_✓ \leftrightarrow v_✗$) if

- both are not pointers, and have the same type, value, and size, or
- both are pointers of the same type and are NULL, or
- both are pointers of the same type and are non-NULL.

Note that two pointers of the same type, but pointing to different addresses, match each other. This is exactly the point of memory graphs: to abstract from concrete addresses.

Matching edges. Two edges $e_✓ = (v_✓, v_✓') \in E_✓$ and $e_✗ = (v_✗, v_✗') \in E_✗$ *match*, written $e_✓ \leftrightarrow e_✗$ if

- the edge expressions are equal,
- $v_✓ \leftrightarrow v_✓'$, and
- $v_✗ \leftrightarrow v_✗'$ — that is, the vertices match.

A.2.4 Computing the Common Subgraph

To compare two memory graphs $G_✓ = (V_✓, E_✓, root_✓)$ and $G_✗ = (V_✗, E_✗, root_✗)$, we use the following *parallel traversal* scheme.

1. Initialize $M = (root_{✓}, root_{✗})$.
2. For all $(v_{✓}, v_{✗}) \in M$, determine the set of *reachable matching vertices* $(v_{✓}', v_{✗}')$ with $v_{✓}' \in V_{✓}$, $v_{✗}' \in V_{✗}$ such that
   - $(v_{✓}', v_{✗}') \notin M$
   - $(v_{✓}, v_{✓}') \in E_{✓}$ (i.e., there is an edge from $v_{✓}$ to $v_{✓}'$)
   - $(v_{✗}, v_{✗}') \in E_{✗}$ (i.e., there is an edge from $v_{✗}$ to $v_{✗}'$)
   - $(v_{✓}, v_{✓}') \leftrightarrow (v_{✗}, v_{✗}')$ (i.e., the edges match, implying $v_{✓}' \leftrightarrow v_{✗}'$)

   Set $M := M \cup (v_{✓}', v_{✗}')$ for each matching pair $(v_{✓}', v_{✗}')$ so found.
3. Continue with step 2 until no further matching vertices can be found.

The matching vertices in M form a common subgraph of $G_{✓}$ and $G_{✗}$. All vertices $v_{✓} \in V_{✓} \cdot (\neg \exists v \cdot (v_{✓}, v) \in M)$ and $v_{✗} \in V_{✗} \cdot (\neg \exists v \cdot (v, v_{✗}) \in M)$ are *nonmatching* vertices and thus form differences between $G_{✓}$ and $G_{✗}$.

Note that M as obtained by parallel traversal is not necessarily the *largest common subgraph*. To obtain this, use the algorithm of Barrow and Burstall (1976), starting from a *correspondence graph* as computed by the algorithm of Bron and Kerbosch (1973).

A.2.5 Computing Graph Differences

We not only need a means of detecting differences in data structures but a means of *applying* these differences. We shall first concentrate on applying *all* differences between $r_{✓}$ and $r_{✗}$ to $r_{✓}$ — that is, we compute debugger commands that change the state of $r_{✓}$ such that eventually its memory graph is identical to $G_{✗}$.

For this purpose, we require three graph traversals. During these steps, $G_{✓}$ is transformed to become equivalent to $G_{✗}$ and each graph operation is translated into debugger commands that perform the equivalent operation on $r_{✓}$.

As an example, we consider the two memory graphs shown in Figure A.4, where the dotted lines indicate the matching M between vertices, obtained from the common subgraph. (Actually, this matching cannot be obtained from parallel traversal, as described in Section A.2.4, but would be obtained from the largest common subgraph.) It is plain to see that element 15 in $G_{✗}$ has no match in $G_{✓}$. Likewise, element 20 in $G_{✓}$ has no match in $G_{✗}$.

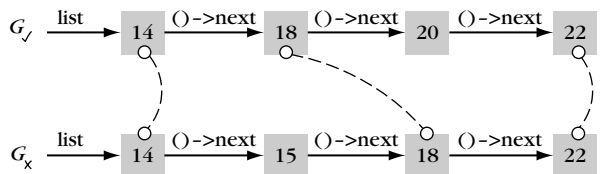

FIGURE A.4

Graph matchings.

1. **(Set and create variables).** *For each vertex v_x in G_x without a matching vertex in $G_✓$, create a new vertex $v_✓$ as a copy of v_x. v_x is matched to $v_✓$.* After this step, each vertex v_x has a matching vertex $v_✓$.

   Figure A.5 shows our example graphs after this step. To generate debugger commands, for each addition of a vertex $v_✓$ we identify the appropriate variable v in r_x and generate a command that

   - creates v in $r_✓$ if it does not exist yet, and
   - sets v to the value found in r_x.

   In our example, we would obtain the following GDB commands.

```
set variable $m1 = (List *)malloc(sizeof(List))
set variable $m1->value = 15
set variable $m1->next = list->next
```

2. **(Adjust pointers).** *For each pointer vertex p_x in G_x, determine the matching vertex $p_✓$ in $G_✓$. Let $*p_x$ and $*p_✓$ be the vertices that p_x and $p_✓$ point to, respectively (reached via the outgoing edge). If $*p_✓$ does not exist, or if $*p_✓$ and $*p_x$ do not match, adjust $p_✓$ such that it points to the matching vertex of $*p_x$.*

   In our example, the *next* pointers from 14 to 18 and from 18 to 20 must be adjusted. The resulting graphs are shown in Figure A.6. Again, any adjustment translates into appropriate debugger commands.

3. **(Delete variables).** *Each remaining vertex $v_✓$ in $G_✓$ that is not matched in G_x must be deleted, including all incoming and outgoing edges.* After this last step, $G_✓$ is equal to G_x.

   In our example, vertex 20 must be deleted. The resulting graphs are shown in Figure A.7.

   Such a deletion of a vertex v translates into debugger commands that set all pointers that point to v to null, such that v becomes unreachable. Additionally, one might want to free the associated dynamic memory.

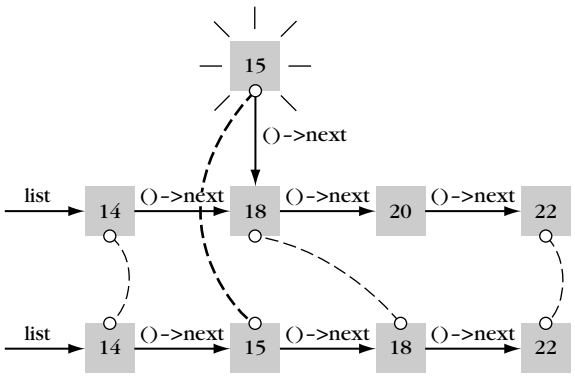

FIGURE A.5

Creating new variables.

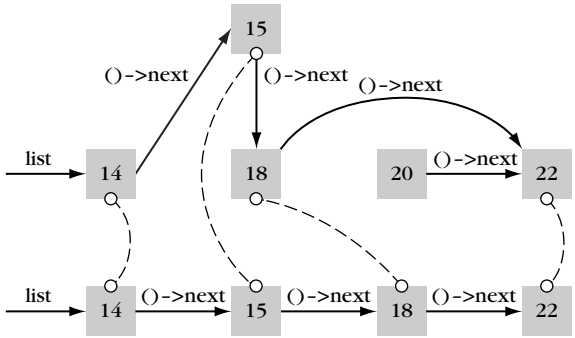

FIGURE A.6

Adjusting pointers.

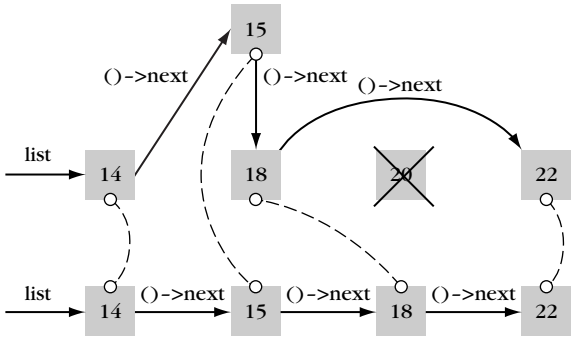

FIGURE A.7

Deleting variables.

After these three steps, we have successfully transferred the changes in a data structure from a run $r_{\text{✗}}$ to a run $r_{\text{✓}}$.

A.2.6 Applying Partial State Changes

For the purpose of delta debugging, transferring *all changes* is not sufficient. We need to apply *partial* state changes as well. For this purpose, we associate a delta δ_v with each vertex v in $G_{\text{✓}}$ or $G_{\text{✗}}$ that is not contained in the matching. If v is in $G_{\text{✓}}$ only, applying δ_v is supposed to delete it from $G_{\text{✓}}$. If v is in $G_{\text{✗}}$ only, applying δ_v must add it to $G_{\text{✓}}$.

Let $c_{\text{✗}}$ be the set of all deltas so obtained. As always, $c_{\text{✓}} = \varnothing$ holds. In Figure A.4, for instance, we would obtain two deltas $c_{\text{✗}} = \{\delta_{15}, \delta_{20}\}$. The idea is that δ_{15} is supposed to add vertex 15 to $G_{\text{✓}}$. δ_{20} should delete vertex 20 from $G_{\text{✓}}$. Applying both δ_{15} and δ_{20} should change $G_{\text{✓}}$ to $G_{\text{✗}}$.

To apply a *subset* $\Delta \subseteq c'_{\times} \setminus c'_{\checkmark}$ only, we run the state transfer method of Section A.2.5, but with the following differences:

- In steps 1 and 3 we generate or delete a vertex v only if δ_v is in Δ.
- In step 2 we adjust a pointer $p_{\checkmark}$ with a matching $p_{\times}$ only if $\delta_{*p_{\times}}$ is in Δ or $\delta_{*p_{\checkmark}}$ is in Δ.

As an example, apply $\Delta = \{\delta_{15}\}$ only. Step 1 generates the new vertex. Step 2 adjusts the pointer from 14 such that it points to 15. However, the pointer from 18 to 20 is not changed, because δ_{20} is not in c. We obtain a graph (and appropriate GDB commands) where only element 15 has been inserted (Figure A.8).

Likewise, if we apply $\Delta = \{\delta_{20}\}$ only step 1 does not generate a new vertex. However, step 2 adjusts the pointer from 18 such that it points to 22, and step 3 properly deletes element 20 from the graph.

A.2.7 Capturing C State

In the programming language C (and its sibling C++), pointer accesses and type conversions are virtually unlimited, which makes extraction of data structures difficult. The following are challenges and how one can deal with them.

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

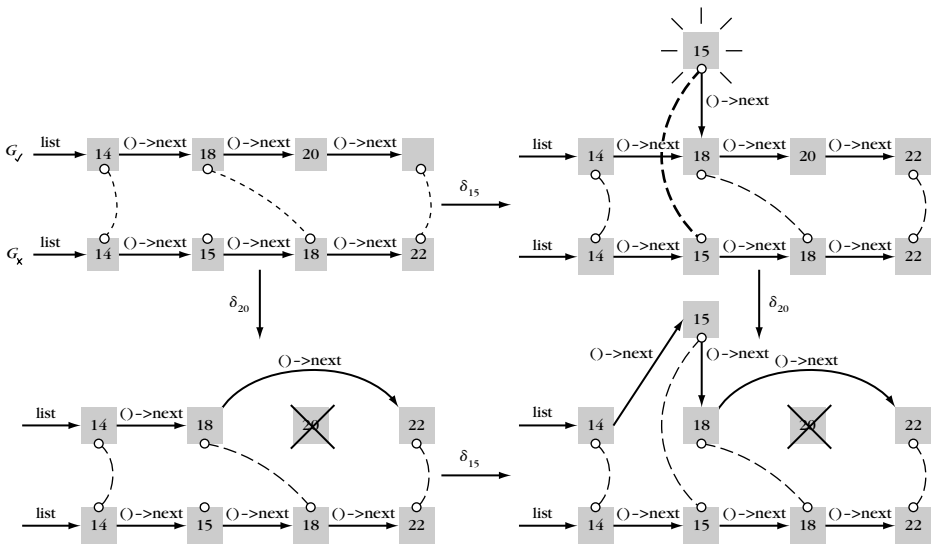

FIGURE A.8

Applying partial state changes.

To distinguish valid from invalid pointers, we use a *memory map.* Using debugger information, we detect individual memory areas such as stack frames, heap areas requested via the *malloc* function, or static memory. A pointer is valid only if it points within a known area.

Dynamic arrays. In C, one can allocate arrays of arbitrary size on the heap via the *malloc* function. Although the base address of the array is typically stored in a pointer, C offers no means of finding out how many elements were actually allocated. Keeping track of the size is left to the discretion of the programmer (and can thus not be inferred by us).

A similar case occurs when a C struct contains arrays that grow beyond its boundaries, as in

```
struct foo {
    int num_elements;
    int array[1];
}.
```

Although `array` is declared to have only one element, it is actually used as a dynamic array, expanding beyond the struct boundaries. Such structs are allocated such that there is sufficient space for both the struct and the desired number of array elements.

To determine the size of a dynamic array, we again use the memory map as described earlier. An array cannot cross the boundaries of its memory area. For instance, if we know the array lies within a memory area of 1,000 bytes the array cannot be longer than 1,000 bytes.

Unions. The biggest obstacle in extracting data structures are C *unions*. Unions (also known as variant records) allow multiple types to be stored at the same memory address. Again, keeping track of the actual type is left to the discretion of the programmer. When extracting data structures, this information is not generally available.

To disambiguate unions, we employ a couple of heuristics, such as expanding the individual union members and checking which alternative contains the smallest number of invalid pointers. Another alternative is to search for a *type tag* — an enumeration type within the enclosing struct the value of which corresponds to the name of a union member. Although such heuristics mostly make good guesses, it is safer to provide explicit disambiguation rules — either handcrafted or inferred from the program.

Strings. A `char` array in C has several uses. It can be used for strings, but is also frequently used as placeholder for other objects. For instance, the *malloc()* function returns a `char` array of the desired size. It may be used for strings, but also for other objects.

Generally, we interpret `char` arrays as strings only if no other type claims the space. Thus, if we have both a `char` array pointer and pointer of another type both pointing to the same area, we use the second pointer for unfolding.

In languages with managed memory such as JAVA or C#, none of these problems exist, as the garbage collector must be able to resolve them at any time. Most languages are far less ambiguous when it comes to interpreting memory contents. In object-oriented languages, for instance, dynamic binding makes the concept of unions obsolete.

A.3 CAUSE–EFFECT CHAINS

A program run r is a sequence of states $r = [s_1, s_2, \ldots, s_n]$. Each state s_i consists of at least a memory graph G_i as well as a backtrace b_i—that is, $s_i = (G_i, b_i)$.

Let $s_\text{✗}$ be a program state from a failing run $r_\text{✗}$. Let $r_\text{✓}$ be a passing run. Then, $s_\text{✓} = match(s_\text{✗})$ is a *matching state*.

Matching states. Two states $s_\text{✓} = (G_\text{✓}, b_\text{✓})$ and $s_\text{✗} = (G_\text{✗}, b_\text{✗})$ match if their backtraces are identical ($b_\text{✓} = b_\text{✗}$). This implies that the set of local variables is equal. The function $match\colon (r_\text{✗} \to r_\text{✓} \cup \{\bot\})$ assigns each state $s_{\text{✗}t} \in r_\text{✗}$ a *matching state* $s_{\text{✓}t} \in r_\text{✓}$, or $\bot$, if no such match can be found.

Individual state differences, as determined by delta debugging, can be composed into a cause–effect chain.

Relevant deltas. For each $s_{\text{✗}t} \in r_\text{✗}$, let a *relevant delta* Δ_t be a failure-inducing difference, as determined by delta debugging: let $s_{\text{✓}t} = match(s_{\text{✗}t})$. If $match(s_{\text{✗}t}) = \bot$ holds, then $\Delta_t = \bot$. Otherwise, let $c_{\text{✗}t}$ be the difference between $s_{\text{✓}t}$ and $s_{\text{✗}t}$, and let $c_{\text{✓}t} = \varnothing$. Let $(c'_{\text{✓}t}, c'_{\text{✗}t}) = dd(c_{\text{✓}t}, c_{\text{✗}t})$. Then, $\Delta_t = c'_{\text{✓}t} \setminus c'_{\text{✗}t}$ is a relevant delta.

Cause–effect chains. A sequence of relevant deltas $C = [\Delta_{t_1}, \Delta_{t_2}, \ldots]$ with $t_i < t_{i+1}$ is called a *cause-effect chain* if each Δ_{t_i} causes the subsequent $\Delta_{t_{i+1}}, \Delta_{t_{i+2}}, \ldots$ as well as the failure.

Within a cause–effect chain, *cause transitions* occur as follows.

Cause transitions. Let $var(\Delta_t)$ be the set of variables affected by a state difference Δ_t. $var(\bot) = \varnothing$ holds. Then, two moments in time (t_1, t_2) are called a *cause transition* if

- $t_1 < t_2$,
- a cause–effect chain C with $[\Delta_{t_1}, \Delta_{t_2}] \subseteq C$ exists, and
- $var(\Delta_{t_1}) \neq var(\Delta_{t_2})$.

A cause transition is called *direct* if $\neg\exists t : t_1 < t < t_2$.

To isolate direct cause transitions, we use a *divide-and-conquer* algorithm. The basic idea is to start with the interval $(1, |r_\text{✗}|)$, reflecting the first and last state of $r_\text{✗}$. If a cause transition has occurred, we examine the state at the middle of the interval and check whether the cause transition has occurred in the first half and/or in the second half. This is continued until all cause transitions are narrowed down.

Isolating cause transitions. For a given cause–effect chain C, the algorithm $cts(t_1, t_2)$ narrows down the cause transitions between the moments in time t_1 and t_2:

$$cts(t_1, t_2) = \begin{cases} \varnothing & \text{if } var(\Delta_{t_1}) = var(\Delta_{t_2}) \\ cts(t_1, t) \cup cts(t, t_2) & \text{if } \exists t : t_1 < t < t_2 \\ \{(t_1, t_2)\} & \text{otherwise} \end{cases}$$

where $[\Delta_{t_1}, \Delta_{t_2}] \subseteq C$ holds.

Our actual implementation computes C (and in particular, Δ_t) on demand. If we isolate a Δ_t between Δ_{t_1} and Δ_{t_2}, but find that Δ_t was not caused by Δ_{t_1}, we recompute Δ_{t_1} such that the cause–effect chain property is preserved.

> We think in generalities, but we live in detail.
>
> – Alfred North Whitehead
> (1861–1947)